THESIS

METHODS TO SECURE DATABASES AGAINST VULNERABILITIES

by

Jonathan P. Sloan

December 2015

Thesis Advisor: Thomas Otani
Second Reader: Mark Gondree

REPORT DOCUMENTATION PAGE		*Form Approved OMB No. 0704–0188*

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** December 2015	**3. REPORT TYPE AND DATES COVERED** Master's thesis
4. TITLE AND SUBTITLE METHODS TO SECURE DATABASES AGAINST VULNERABILITIES		**5. FUNDING NUMBERS**
6. AUTHOR(S) Jonathan P. Sloan		
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000		**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A		**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol number ____N/A____.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited	**12b. DISTRIBUTION CODE**

13. ABSTRACT (maximum 200 words)

Many commercial and government organizations utilize some form of proprietary or open source database management system. Recent history shows security incidents involving database management system vulnerabilities resulting in the compromise of personal information for millions of people. This thesis identifies common vulnerabilities affecting database management systems: injection, misconfigured databases, HTTP interfaces, encryption, and authentication and authorization. This thesis also examines three open source database management systems: MySQL, MongoDB, and Cassandra. We test each against the aforementioned vulnerabilities and provide recommendations to mitigate the vulnerabilities.

14. SUBJECT TERMS database, security, injection, encryption, authentication, authorization, MySQL, MongoDB, Cassandra			**15. NUMBER OF PAGES** 89
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UU

NSN 7540–01-280-5500 Standard Form 298 (Rev. 2–89)
Prescribed by ANSI Std. 239–18

THIS PAGE INTENTIONALLY LEFT BLANK

METHODS TO SECURE DATABASES AGAINST VULNERABILITIES

Jonathan P. Sloan
Lieutenant Colonel, United States Army
B.S., United States Military Academy, 1997

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December 2015

Author:	Jonathan P. Sloan
Approved by:	Thomas Otani, Ph.D. Thesis Advisor
	Mark Gondree, Ph.D. Second Reader
	Peter Denning, Ph.D. Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Many commercial and government organizations utilize some form of proprietary or open source database management system. Recent history shows security incidents involving database management system vulnerabilities resulting in the compromise of personal information for millions of people. This thesis identifies common vulnerabilities affecting database management systems: injection, misconfigured databases, HTTP interfaces, encryption, and authentication and authorization. This thesis also examines three open source database management systems: MySQL, MongoDB, and Cassandra. We test each against the aforementioned vulnerabilities and provide recommendations to mitigate the vulnerabilities.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

ACID	Atomicity, Consistency, Isolation, Durability
ANSI	American National Standards Institute
API	Application Programming Interface
BSON	Binary JSON
CQL	Cassandra Query Language
CWE	Common Weakness Enumeration
DOD	Department of Defense
GUI	Graphic User Interface
ISO	International Organization for Standardization
JSON	Java Script Object Notation
NoSQL	Not Only SQL
NCDBMS	NoSQL Column-Family Database Management System
NDDBMS	NoSQL Document Database Management System
OWASP	Open Web Application Security Project
RDBMS	Relational Database Management System
SQL	Structured Query Language

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I would like to thank Professor Thomas Otani for all the guidance, and support that he provided to me in completing this thesis. He enabled me to focus on clearly identifying the scope of the thesis, and he provided valuable insight on scholarly writing.

I would also like to thank Professor Mark Gondree for his invaluable assistance as second reader.

Finally, I would like to thank my lovely wife, Denise, for all her support throughout my time at the Naval Postgraduate School. Without her understanding, I would not have been able to dedicate the amount of time required to complete this thesis and maximize this learning experience.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

Databases have been in existence for over 40 years. Many organizations operate one or more databases that contain customer information, sales numbers, weapons inventory, or even intelligence documents. The data is normally kept for storage and retrieval, and analysis [1]. Currently, there are two broad categories of database models, relational and NoSQL.

A relational database consists of tables where each table defines an entity with each column specifying an attribute of that entity [1]. Each row is an instance of the entity, and while these instances are separate, they are all similar in they are all defined by the same attributes. Not surprisingly, relational databases work best with data that has a similar structure [1].

The NoSQL model (or Not Only SQL), on the other hand, is not built upon a collection of related tables. It is designed more for data that is not easily represented by the relational model [2]. The NoSQL model also serves large-scale datasets that can be distributed across multiple nodes with parallel data processing [2]. This thesis is concerned with identifying the security vulnerabilities present in both database systems, and how to mitigate them.

A. MOTIVATION

Most U.S. government organizations use some type of database. The National Security Agency created a NoSQL database called Accumulo that it uses for data analysis, which is now an open source database [3]. The U.S. Department of Veterans Affairs uses the open source relational database MySQL to store and manage data [4]. The Department of Defense (DOD) Office of the Inspector General also uses a MySQL database to organize and store reports, and provides a web interface where users can query the database and receive information [5].

These are just three government-agency examples. One commonality among these three is that each database system is open source as opposed to proprietary. In the past, the DOD mainly used proprietary systems across myriad technologies. However, the

DOD has been moving toward open source systems in order to lower costs, acquire and repair capabilities, and share information [6]. While there are many benefits to using open source technology, one security drawback is the code is available to everyone. An attacker does not need to expend resources to acquire the code in order to study and test it for vulnerabilities. It is also more advantageous for an attacker to have the source code than having only a binary. An attacker who has access to a binary only must commit time to dumping the binary and unravelling the assembly instructions. An attacker with the source code can use this as a guide when analyzing the binary to determine potential avenues for exploitation.

B. DATABASE VULNERABILITIES AND BREACHES

The Open Web Application Security Project (OWASP) first released its Top 10 Project list in 2003, identifying the most critical application security risks. It updated the list in 2007, 2010, and most recently, in 2013. Three of the risks in the 2013 OWASP Top 10 can be found in database systems: “injection” at number 1, “security misconfiguration” at number 5, and “sensitive data exposure” at number 6 [7].

The Common Weakness Enumeration (CWE)/SANS Institute publishes a top 25 list of security vulnerabilities with its most recent version in 2011. Five of the vulnerabilities in its list are also found in database systems: “Improper neutralization of special elements used in a SQL command” at number 1, “Missing authentication for critical function” at number 5, “Missing authorization” at number 6, “Missing encryption of sensitive data” at number 8, and “Reliance on untrusted inputs in a security decision at number 10 [8]. These vulnerabilities are discussed in this thesis.

Underscoring these vulnerabilities are the numerous breaches into government and company databases leading to large losses of data. The following are just a few organizations that suffered database breaches over the past five years. In 2011, the email marketing giant Epsilon suffered a data breach, and customer emails from 50 major clients (such as Kroger and U.S. Bank) were stolen [9]. Also in 2011, Sony PlayStation Network suffered a data breach that affected 70 million users [10]. A breach of JP Morgan Chase in 2014 resulted in the theft of data from 76 million households and 7 million businesses [11]. Also in 2014, it was revealed that 22.8 million personal records of New York residents were stolen over eight years during 5000 separate incidents [12]. In 2014, Sony suffered another data breach resulting in the theft of intellectual property, emails, and information on 15,000 employees [13]. This year, Anthem, Inc., the second largest health insurer in the United States, experienced a data breach that affected tens of millions customers [14]. In addition, this year, the Office of Personnel Management suffered a data breach resulting in the theft of 21.5 million federal employees' records [15].

C. THESIS ORGANIZATION

Chapter II focuses on the background of three database applications that are representative of the relational database model, the NoSQL document model, and the NoSQL column family model. The chapter covers the design and query language of MySQL (relational), MongoDB (NoSQL document), and Cassandra (NoSQL column family). Chapter III provides a brief overview of security vulnerabilities (injection, misconfigured databases, HTTP interface, encryption, authentication and authorization) in databases, and then describes the methodology we used to examine each database system against these vulnerabilities. Chapter IV, V, VI, VII, and VIII describe in more detail the security vulnerabilities, show whether the applications are affected by the vulnerabilities, and detail the mitigations against these vulnerabilities. Chapter IX covers the thesis conclusion.

THIS PAGE INTENTIONALLY LEFT BLANK

II. BACKGROUND

This chapter first briefly reviews three database models: relational, NoSQL document, and NoSQL column-family. Then we examine in depth three database management systems fitting these models: MySQL, MongoDB, and Cassandra respectively. These three database management systems will run on many different operating systems, but for this thesis, we only consider the Microsoft Windows family of operating systems.

A. DATABASE MODELS

This section provides a brief overview of three database models: relational, NoSQL document, and NoSQL column-family.

1. Relational Database Model

The core element of a relational database is a table [1]. A relational database table models some entity, for example, an employee. The columns of the table describe common attributes that all employees share, for example name, address, and salary [1]. There could be more or less attributes, but the important characteristic is that all employees share these attributes. One of the attributes is chosen as the primary key. This important attribute uniquely identifies a specific instance of the entity, which is a row in the table [1]. In this example, the primary key may be the employee's identification number.

As the name implies, entities in the relational database model can have relationships with each other [1]. For example, an employee, which is one entity, can work for a department, which is another entity. These relationships may also be represented by tables, and again each column would represent an attribute that all relationships have in common [1]. Therefore, in this database model, all data is very structured.

These relationships could be one-to-many (a department has many employees), one-to-one (an employee may have one cubicle), or many-to-many (one employee can talk to many customers and one customer can talk to many employees). These relationship cardinalities and additional rules governing the relationships are what the database management system uses to enforce consistency within the database [16].

Consistency is a key component of ACID (Atomicity, Consistency, Isolation, and Durability), which is what Relational Database Management Systems (RDBMS) provide to those who choose to use the relational database model [16]. Atomicity means that a RDBMS will complete all elements of a transaction or will complete no elements of the transaction in the event of an error [16]. In this case, the RDBMS will try again to complete the entire transaction. Consistency means that a RDBMS will ensure that after every write to the database, any subsequent read to the database will reflect the new data and not the old data [16]. Isolation means the RDBMS will ensure that individual transactions will not interfere with other transactions [16]. Durability means the RDBMS will ensure that any changes to the database are persistent [16]. A write to the database will not suddenly roll back to a previous state. Examples of RDBMS are MySQL, Microsoft Access, and Microsoft SQL Server.

2. NoSQL Document Database Model

The core element of a NoSQL document database model is a document [1]. Documents are then grouped in collections [1]. Documents and collections are loosely analogous to a rows and tables in the relational database model. In this model, however, individual documents within a collection need not share common attributes with other documents [1]. Continuing the example from above, a collection may describe employees. In this case, one document can represent an employee who has three attributes, and another document can represent an employee who has nine attributes.

Javascript Object Notation (JSON) and Binary JSON (BSON) are two standardized formats that a NoSQL Document Database Management System (NDDBMS) may use to represent documents [2]. JSON and BSON are used to form the internal elements of a document. A document is comprised of one or more key/value pairs where the key and value are separated by a colon [2]. Key/value pairs are further separated by commas and documents are surrounded by brackets [2]. An example would be {name:”John,” age:33}. A value can be a variety of datatypes to include strings, integers, arrays, and even other documents [2]. An example would be {name:{first:”John,” last:”Doe”}, age:33, dogs: [“fido,” “rover”]} where the first value is a document, the second value is an integer, and the third value is an array of strings.

A document may contain unstructured, semi-structured, or structured data [2]. An example of semi-structured data is one where many employees share some common attributes, but also may have some attributes that other employees do not have. A relational database model would require a table for employees where many columns represent every single attribute that all employees have whether or not they are shared. The RDBMS would then store null bytes in rows where employees do not have an attribute [2]. In the NoSQL document database model, however, an employee with three attributes would have three key/values pairs and an employee with nine attributes would have nine key/value pairs. This is referred to as sparse data [2]. Examples or NDDBMS are MongoDB and CouchDB.

3. NoSQL Column-Family Database Model

The core element of a NoSQL column-family database model is a table [1]. The tables in this model differ from the tables in the relationship database model, however. In this model, a table is a mixture of a NoSQL document database collection and a relational database table [1]. Tables represent entities and columns represent the attributes of those entities. Like the relational database model, the total number of columns in a table would equal the total number of distinct attributes that each instance of the entity has. Rows represent specific instances of an entity. In the relational database model where rows have uniformed fixed sizes, a null byte is required in a column where an entity instance does not have the attribute represented by the column [2].

In the NoSQL column-family database model, however, there is no requirement that rows are a uniformed fixed size [1]. A NoSQL column-family database management system (NCDBMS) would not store a null byte to represent that an entity instance does not have the attribute represented by the column [2]. The row would only consist of those values that the entity instance actually has. This characteristic of a row in a NoSQL column-family database models is more akin to how a document looks in a NoSQL document database model [2].

This model further differentiates itself from the relational database model in that the value in a row need not be a single value [2]. For example, a table in this model may represent the employee entity. One of the columns may be “phone numbers.” In this model, one row may have an array of three phone numbers, another row may have an array of one phone number, and yet another row may not have a phone number at all. This aspect of the NoSQL column-family database model is similar to the NoSQL document database model [2]. Examples of NCDBMS are Cassandra and BigTable.

B. MYSQL

This section examines the RDBMS MySQL in depth. The organization of this section is MySQL overview, installation and configuration defaults, accounts, security, query language, and stored procedures.

1. MySQL Overview

MySQL is an open source RDBMS written in C and C++ and made available by Oracle Corporation [16]. Open source means that any organization may use the software free of charge and modify the software to fit the needs of the organization as long as the modifications do not violate the General Public License [16]. MySQL will run on several operating systems to include Microsoft Windows, Macintosh OSX, and Linux [16].

MySQL is designed to work in a client/server configuration [16]. The server portion of MySQL will work on both desktop and laptop computers [16]. Additionally, it can exist on only one machine or a network of clustered machines all running MySQL [16]. Communication with the server can take the form of a MySQL provided client shell, the GUI MySQL Workbench, or a third party program employing application programing interfaces (API) for several languages such as C, C++, PHP, Java and Python [16].

MySQL will work well with very large databases. The documentation references anecdotal evidence of MySQL supporting databases with 50 million records, 200,000 tables, and 5,000,000,000 rows [16]. It also supports a variety of datatypes. Users can employ signed and unsigned integers, floats, doubles, characters, strings, binary, text, binary large objects, and various date and time representations [16].

2. Installation and Configuration Defaults

MySQL provides an installer that allows the user to choose the setup. The recommended setup is Developer, which will install MySQL server, MySQL Workbench, and some other tools [16]. This setup will also provide an option to create a Windows service for MySQL server so that it will start up automatically when the computer restarts [16].

The installer gives the user the option of creating accounts that have permissions such as administrator and designer. It also gives the option of creating a user MysqlSys that would run the MySQL server with limited privileges [16]. None of these is mandatory. The only mandatory action regarding users is creating a root account and assigning a password for the root user [16]. This is an improvement over previous versions of MySQL where a password was not required when creating the root account [16].

Several defaults can be changed during the configuration process that the user should be aware of. One default is that TCP/IP networking is enabled [16]. This allows remote systems to connect to the database. Disabling this feature will allow only localhost connections to the database [16]. Another consequence of the default setting is that the default IPv4 bind address is 0.0.0.0. This means that the MySQL server will attempt to bind to all network interfaces. The result of this setting could be an increase in the number of entry points into the database.

MySQL maintains a table called mysql.user, which now has a column called “password_expired” [16]. The default value of this column entry is “N,” but can be changed to “Y” for individual users [16]. This allows the database administrator to require users to change passwords regularly. Another relevant security issue is that by default there is no requirement for encrypted connections [16].

3. Accounts

As discussed above, MySQL installation requires the creation of a root account with a password. MySQL will actually require two root accounts if the user selects the default option of TCP/IP connections [16]. One root account will handle localhost connections and the other root account will handle remote connections [16]. This initial password requirement, however, does not prevent a root user from changing the root password to the empty string later. Since these accounts are superusers, great care must be taken to ensure these accounts have very good passwords.

The root user or another user with equivalent privileges can create additional users and grant specific privileges to each user [16]. Again, great care should be shown in granting privileges to users. An attacker who compromises a user account will have all privileges that have been granted to that user. One other special account is “anonymous.” The installer can optionally create anonymous accounts. These allow connection to the database with no username or password [16]. Database administrators should remove these accounts if they are not necessary.

4. Security

This subsection examines security aspects of MySQL including authentication and authorization, encryption, and other issues.

a. Authentication and Authorization

Authentication is a security term that deals with allowing a specific user to login with some piece of information that confirms the identity of the user. Authorization refers to the privileges that have been granted to a specific user. MySQL uses authentication and authorization by default [16]. MySQL has commands for both creating users (and requiring passwords) and granting specific privileges to users [16]. By default, newly created users have no privileges at all [16]. Such users may still be able to access databases that provide generic privileges to any user (such as the “test” database) [16].

An administrator can assign to users privileges to execute specific commands [16]. These commands include, but are not limited to, SELECT, INSERT, UPDATE, and

DELETE [16]. The administrator can grant these privileges to users for specific databases or to all databases [16]. Additionally, the administrator, or anyone with the GRANT privilege for a specific database, can assign to users privileges for specific database objects like tables, views, and stored procedures [16]. One caveat is that a user who has the privilege to create or drop tables in a specific database will also have the privilege to create or drop the database itself. MySQL does not allow separation of these two privileges [16].

There are many special administrative databases that MySQL provides, but two databases of note are the "information_schema" and "performance_schema" databases. Every user has access to the "information_schema" database but will see only information in the tables of this database that corresponds to database objects that the user has privileges to see [16]. If a user has the SELECT privilege for every database and the user's account is compromised by an attacker, the attacker will be able to learn every database name, all table names, and all column names. The "performance_schema" database by default is only accessible by the root account, but the root user can grant to a user access to this database [16]. A user with access to this database can learn user names and account names. Since user names and account names are typically repeated for other network services, it is important that access to this database is not granted lightly.

b. Encryption

MySQL does not enable encryption of communications between the client and server by default [16]. However, MySQL does support SSL connections for client-server communications [16]. MySQL does protect passwords for storage by default using SHA-256 [16]. MySQL also offers built-in functions for encrypting data for storage in the database.

c. Other

MySQL allows the administrator to change content of error messages [16]. It is important to minimize the information that a database system provides by way of error messages. Attackers may intentionally try to induce an error in order to receive an error message that reveals information about the database.

MySQL does not appear to offer any built-in system stored procedures. System stored procedures can be an avenue that an attacker may use to remotely take over a computer through an attack on the database system. Chris Anley [17] showed how an attacker could execute arbitrary command lines using the xp_cmdshell stored procedure in Microsoft SQL Server 2000 by way of SQL injection.

5. Query Language

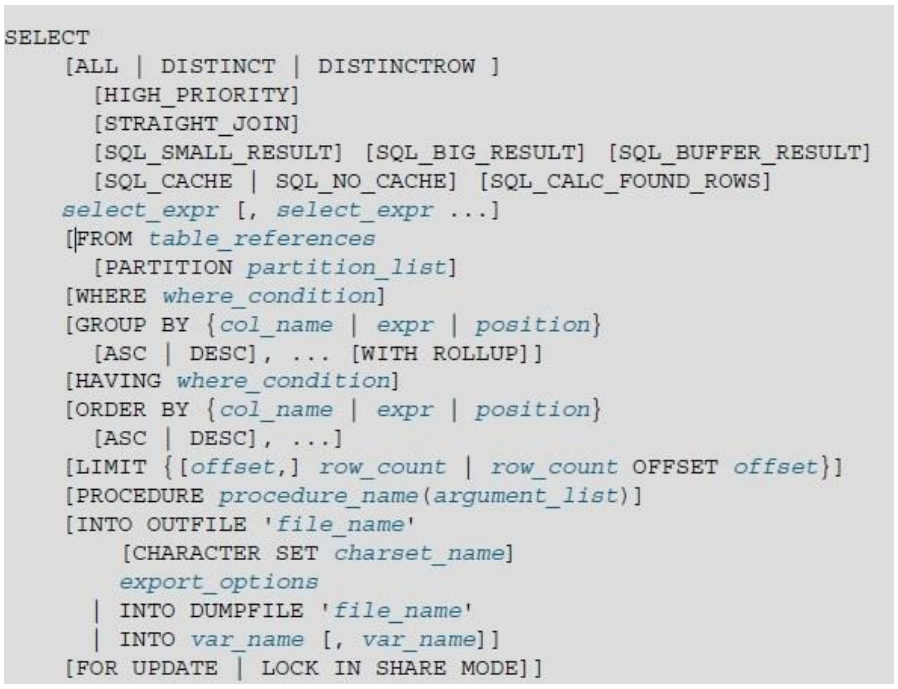
Like any other relational database, MySQL uses the Structured Query Language (SQL) for accessing the database [16]. Unlike the various query languages used by NoSQL database systems, there are ANSI/ISO standards for SQL [16]. ANSI/ISO have updated the standard over the years so several versions exist [16]. MySQL incorporates several different versions of SQL into the query language used for its product [16]. The user manual explicitly refers to the standard version (i.e. SQL-92, SQL:1999, etc.) when describing the use of a particular statement [16]. This section provides a brief overview of database and table-altering commands, and then examines the SELECT statement.

a. Database and Table Altering Commands

MySQL provides several different SQL commands for altering the database and tables within databases [16]. These commands are CREATE, DROP, and ALTER [16]. The database administrator can grant privileges for each command for each database [16]. CREATE allows a user to create databases, tables, and other database objects [16]. The DROP command allows a user to delete a database, table and other database objects [16]. This is a very dangerous command as there are not any warnings attached to its use (i.e. “do you really want to drop the database?”). Administrators should be judicious in granting the privilege for this command. If an attacker compromises an account that has this privilege, the attacker can cause grave damage. The ALTER command allows a user to change a database, table, or other database object [16].

b. SELECT Statement

The SELECT statement is the primary command used to query the database for information [16]. Figure 1 depicts the syntax of the SELECT statement.



```
SELECT
    [ALL | DISTINCT | DISTINCTROW ]
      [HIGH_PRIORITY]
      [STRAIGHT_JOIN]
      [SQL_SMALL_RESULT] [SQL_BIG_RESULT] [SQL_BUFFER_RESULT]
      [SQL_CACHE | SQL_NO_CACHE] [SQL_CALC_FOUND_ROWS]
    select_expr [, select_expr ...]
    [FROM table_references
      [PARTITION partition_list]
    [WHERE where_condition]
    [GROUP BY {col_name | expr | position}
      [ASC | DESC], ... [WITH ROLLUP]]
    [HAVING where_condition]
    [ORDER BY {col_name | expr | position}
      [ASC | DESC], ...]
    [LIMIT {[offset,] row_count | row_count OFFSET offset}]
    [PROCEDURE procedure_name(argument_list)]
    [INTO OUTFILE 'file_name'
        [CHARACTER SET charset_name]
        export_options
      | INTO DUMPFILE 'file_name'
      | INTO var_name [, var_name]]
    [FOR UPDATE | LOCK IN SHARE MODE]]
```

Figure 1. SELECT statement syntax, from [16].

One can use the SELECT statement to fetch rows from one table or multiple tables [16]. Selecting from multiple tables is also known as a join and is unique to relational databases [16]. The “select_expr” in Figure 1 is used to narrow down the columns that the SELECT statement will return [16]. A user can select one table, multiple tables, or “*” for all tables [16]. The “table_references” in Figure 1 indicates the tables the user wants to query [16]. A user can query both tables in the current database and tables in other existing databases in one SELECT statement [16]. SELECT and FROM are the only mandatory portions of the SELECT statement [16].

A user can refine the SELECT statement by employing the WHERE clause to stipulate criteria that rows must possess in order to be returned by the SELECT statement [16]. The user provides the optional “where_condition” that is then evaluated row by row in the tables indicated in the FROM clause. Rows that evaluate to true are included in the return set [16].

Unions and subqueries are used in conjunction with the SELECT statement [16]. A user can employ the UNION statement in order to merge the results from two or more

SELECT statements [16]. The column names in the first SELECT statement will be used as the column names for the merged returned result [16]. A subquery on the other hand is a complete SELECT statement that is nested within a SELECT statement [16].

6. Stored Procedures and Triggers

Stored procedures and triggers are both database objects that are written in SQL [16]. A user creates a stored procedure to perform a function or query that occurs frequently. For example, a user may wish to query the number of employees in a department. Instead of repeatedly typing the SELECT statement, the user can type the SELECT statement once when creating the stored procedure and then call the stored procedure to process the query. A stored procedure can take zero, one, or more arguments [16]. An important security note regarding stored procedures is that it executes with the privileges of the user who defined it [16]. An attacker who compromises a stored procedure defined by the root account may be able to perform actions with root privileges. Triggers are used to perform some action once some event occurs, typically an insert or update [16]. These can be used to enforce some rule defined in the entity-relationship diagram that models the database.

C. MONGODB

This section examines the NDDBMS MongoDB in depth. The organization of this section is MongoDB overview, data modeling, installation and configuration defaults, security, and query language.

1. MongoDB Overview

MongoDB is an open source NDDBMS that was developed by the company 10gen, now referred to as MongoDB, Inc. [18]. MongoDB maintains databases made up of collections [19]. A collection is similar to a table in MySQL, but collections do not require a defined schema [20]. Collections are made up of documents that are analogous to the rows in a MySQL table [20]. Documents are made up of one or more BSON key and value pairs [21].

The key (or field) name in the BSON key and value pair is a string [21]. There will always be at least one key and value pair [21]. If the user does not create the "_id" key, then MongoDB will create this key and assign a unique value to it [21]. This key will always be the first key in the document and is somewhat analogous to the primary key in a relational database table [21]. The value for the "_id" field must be unique in the collection. If the user chooses to provide the value for the "_id" field, then the user may select any BSON data type except for an array [21]. The BSON data types available in MongoDB are double, string, object (other documents), array, binary data, ObjectID, Boolean, date, null, regular expression, JavaScript, and integer (32-bit and 64-bit) [21].

Like many NoSQL databases, MongoDB provides both replication and sharding. Replication is reproducing the same data on different servers in order to ensure redundancy and provide greater availability of data [21]. This is done through the use of replica sets [21]. Since there are multiple copies of the same data, consistency can become an issue when reading directly after a write because the write may not have replicated to all the replica sets yet [21]. MongoDB provides eventual consistency in favor of greater availability [21]. Sharding is the partitioning of documents from one collection on multiple machines, and is useful in growing a database beyond the size limitations of one machine [21].

MongoDB provides a command line client for communicating with the server [21]. However, there are also several APIs that a user may employ to author their own client programs. APIs are available for several languages such as C, C++, Java, Python, and PHP [21].

2. Data Modeling

MongoDB does not require a rigid schema like MySQL [21]. A user can create a collection and insert a document without defining any attributes for the collection. It is important to define some structure for a collection to aid in querying the collection, but the non-rigid schema allows some documents to have more or less attributes than other documents in the collection.

MongoDB provides for two methods to model relationships between entities. The first is references, and the second is embedded documents [21]. Using references is referred to as a normalized data model [21]. One document in a collection will refer to a document in another collection by including a key and value pair where the value is the "_id" for the first document [21]. For example, one document in the employee collection may refer to a document in the department collection in order to depict that an employee works in a certain department. The key would be "DepartmentID" and the value would be the "_id" for the department document. In this model the document in the department collection only needs to be created once and then all employees who work for that department will have a key and value pair referencing the document in the department collection. While this does save space, two queries are required to discover the name of the department where the employee works: one query of the employee collection to retrieve the referenced key and value pair, and one query of the department collection to retrieve the name of the department. The reason for this is MongoDB does not support joins [21]. Additionally, MongoDB only supports atomicity at the single document level [21]. Updating both employee information and department information at the same time will not be an atomic operation.

Using embedded documents is referred to as a denormalized data model [21]. Using the same example as above, a document in the employee collection will have a key and value pair where the key is "department" and the value is an embedded document containing all the department information. In this model, there is only one query required to discover the name of the department where the employee works. Additionally, updating both the employee information and department information at the same time will be an atomic operation since the department information is embedded in the employee document [21]. However, the department information is duplicated for every employee who works in the department. Updating the name of the department will require updating the document for every employee who works for that department. The database designer may use one or both models to model various relationships.

3. Installation and Configuration Defaults

Installation is very simple, but there is no graphical user interface as with the MySQL installation. MongoDB will not start as a Windows service by default [21]. The user documentation explains how to start MongoDB as a Windows service [21]. Installation provides the server programs for both single node (mongod.exe) and sharded nodes (mongos.exe), and the client shell for communicating with the server (mongo.exe) [21].

The MongoDB server by default will attempt to bind to all available network connections (0.0.0.0) [21]. By default, MongoDB does not require authentication or authorization [21]. This means that anyone could connect to a newly installed MongoDB database simply by knowing the IP address of the server. There is also no encryption of client to server communication by default [21]. MongoDB provides a HTTP interface, but it is disabled by default [21]. If enabled, this interface is read-only by default [21].

4. Security

This subsection examines security aspects of MongoDB including authentication, authorization, encryption, and other issues.

a. Authentication

Authentication is not enabled in MongoDB by default, but can be enabled in the configuration file [21]. MongoDB offers three different ways to authenticate clients. The first is a challenge and response protocol using SCRAM-SHA-1 [21]. This is the default when authentication is enabled [21]. The remaining authentication mechanisms are via x.509 certificates, LDAP proxy authentication, and Kerberos authentication [21]. The database administrator has two options to enable authentication. The first is to startup MongoDB without authentication, create an administrator account and password, shutdown MongoDB, enable authentication in the configuration file, and then restart MongoDB [21]. The second is to enable authentication before startup with localhost exception [21]. This will enable the administrator to establish a connection to the database from the localhost and create an administrator account and password [21]. The exception only applies when no users have been created yet, and the only action the administrator can take is to create the first account [21]. The administrator would then log out, reconnect and authenticate with the newly created administrator account, and then create other user accounts [21].

MongoDB also provides an option for requiring a machine to authenticate into a replica set or sharded cluster [21]. This option is disabled by default [21].

b. Authorization

Authorization is not enabled in MongoDB by default, but can be enabled in the configuration file [21]. Enabling authorization will by default also enable authentication [21]. Authorization enables MongoDB to control what a user can access through the roles that the administrator assigns to the user [21]. This is referred to as Role-Based Assess Control [21]. A role in this instance is defined as a set of privileges and privilege is defined as a set of actions and resources where the user is allowed to take the actions upon the resources [21].

c. Encryption

Encryption of client-to-server communications and vice versa is not enabled in MongoDB by default [21]. The database administrator has the option of using TLS/SSL to encrypt the client and server communications [21]. There does not appear to be any built-in functions in MongoDB for encrypting data for storage. A user who wishes to encrypt data prior to storage will need to encrypt the data with a separate application and then insert the data into the database.

d. Other

MongoDB has a web interface that is not enabled by default [21]. This interface provides diagnostic and monitoring information [21]. When enabled the interface only provides this information by default, but the administrator has the option of allowing built-in queries through the interface [21]. By default, the port is 1000 greater than the port upon which the server accepts connections [21]. This cannot be changed in the configuration file. It is important to note that the web interface does not support authentication [21]. If enabled and the default bind address (0.0.0.0) is used, then anyone would be able to access the web interface.

5. Query Language

Unlike SQL in RDBMS, there are no standard query languages for the various NoSQL database types (i.e., document, column-family, etc.). MongoDB provides its own query language that is distinct from other NDDBMS [21]. The MongoDB's language provides commands to perform many actions that are similar to tasks available in SQL. MongoDB supports the creation and deletion of collections, insertion and deletion of documents, etc. [21]. This thesis focuses on the command used to search for information.

MongoDB uses the db.collection.find() method to search for information where "collection" is the name of the collection the user would like to query [21]. In this section, we consider a collection named "employee." If a user wants all the documents in the employee collection, then the user would provide no argument to the method resulting in db.employee.find(). A user can optionally provide an argument to the find method that must be in BSON format [21]. An example would be db.employee.find({name:{first:"john," last:"doe"}}).

A user can specify an argument that supports an equality match. The argument would be a document of the form {<key>:<value>} where <value> is the data the user wants to match [21]. An example would be {name:"john"}. The example in the preceding paragraph uses an embedded document for the key "name." MongoDB allows dot notation in order to use only one portion of the embedded document in the query [21]. An example would be db.employee.find({"name.last":"doe"}) where the user only wants to search by last name. MongoDB also supports non-equality matches such as greater than, less than, and not equal [21]. A "greater than" example would be db.employee.find({age:{$gt:40}}) to find all employees who are older than 40. The <value> could also be an array of values. A user can specify in the search for the entire array to be present, one element to be present, an element to be present in a specific position, or that an element meets a specific condition (i.e., less than 10) [21].

A successful query returns all the documents that meet the condition of the query [21]. All the keys will be included in the document by default to include the "_id" key [21]. The user can specify keys by including a projection with the find method, which is also in BSON format [21]. An example would be db.employee.find({age:40}, {name:1, _id:0}) to return all employees who are 40 where the only key displayed is name. Unlike MySQL, MongoDB does not support joins, therefore a user can only query one collection per query with the find method.

D. CASSANDRA

This section examines the NCDBMS Cassandra in depth. The organization of this section is Cassandra overview, data modeling, installation and configuration defaults, security, and query language.

1. Cassandra Overview

Cassandra is an open source NCDBMS that was originally designed by Facebook, Inc. to support user searches of their Inbox [19]. Cassandra was able to support the millions of users using Facebook's Inbox search application at any one moment [19]. Much like MongoDB, Cassandra enables the distribution and replication of data amongst a large number of servers so that data is highly available [19].

Cassandra maintains keyspaces, which are analogous to databases in MySQL and MongoDB [22]. Keyspaces contain tables, and tables are made up of partitioned rows [22]. A table is loosely similar to a table in MySQL where a table represents an entity, and rows represent instances of that entity. Each row has a primary key and zero or more clustering keys [22]. Cassandra, however does not require a rigid schema [23]. Entity instances may share attributes, but this is not required. Furthermore, entity instances may have more or less attributes than another entity instance. The rows of a table are not required to exist on the same machine, but rather can be partitioned amongst several nodes using the primary key [23]. Cassandra's partitioning and replication allows databases to scale linearly much like MongoDB [23].

Users who wish to deploy a Cassandra server can get the software package from Apache for use on Linux operating systems [24]. Apache also provides the source code [24]. Users can get software packages from Datastax for use on Linux, Microsoft Windows, and Macintosh OSX, but does not provide the source code [24]. Both Apache and Datastax provide a Cassandra Query Language (CQL) shell for client communication with the server [24]. Datastax also provides C#, Java, and Python drivers so that users can create their own client programs for accessing the server [25].

Cassandra also provides replication and sharding like MongoDB [22]. Users that wish to use the replication capability would attach nodes to a Cassandra ring and customize the amount of replication they would like [22]. Users can add nodes to scale their databases horizontally [22]. Cassandra is also able to automatically distribute data across the nodes in the Cassandra ring to shard the database [22]. Clients who wish to read or write from the database contact any node in the ring, and the node coordinates the request operation with the nodes in the ring that contain the data [22].

2. Data Modeling

Cassandra models entities as tables of partitioned rows with each row representing an instance of the entity [25]. Each row has either a simple primary key (the first column in the table) or a compound primary key (the first two or more columns in the table) [25]. The first column is also known as the partitioning key as it determines upon which node Cassandra will store the row's data [25]. The remaining columns (known as clustering columns) in a compound primary key determine how the rows are sorted on a node [25].

Cassandra's data model is based on denormalization [25]. Relationship between entities are not represented as additional tables as in MySQL, but within the entity tables themselves as collections. Collections in Cassandra consist of three datatypes: set (collection of unique values), list (collection of ordered values), and map (collection of name and value pairs) [25]. For example, an employee table may have a column called department that is of datatype map. The name and value pairs could be {“name”:”human relations”}. This data would then be repeated amongst all employees in the human relations department, and there may even be a table called human relations that provides information about the department.

3. Installation and Configuration Defaults

Cassandra is available in Linux, Microsoft Windows, and Macintosh OSX. The Windows version is only available through Datastax and is titled Datastax Community [22]. This software contains the Cassandra server and other tools such as the CQL shell and OpsCenter, a web based access tool for diagnostics and administrative information [22]. The installer provides a wizard that allows the user to decide whether or not to set up Windows services for the Cassandra server and OpsCenter.

Cassandra has a configuration file called “cassandra.yaml” for establishing any desired settings. By default, Cassandra will only bind to the localhost (127.0.0.1) and use ports 9042 for CQL and 9160 for the Thrift service [22]. Both CQL and Thrift are enabled by default [22]. There is no authentication or authorization by default, nor is there any encryption by default for client-to-server communications [22]. There is also no authentication or encryption by default for internode communication [22]. There is a separate configuration file for the OpsCenter called “opscenterd.conf.” By default, the OpsCenter binds to all available network connections (0.0.0.0) and listens to port 8888 [26].

4. Security

This subsection examines security aspects of Cassandra including authentication and authorization, encryption, and other issues.

a. Authentication / Authorization

Both authentication and authorization are disabled in Cassandra by default [22]. Both of these can be enabled through the configuration file. After enabling authentication the default userid and password is cassandra/cassandra [22]. The user cassandra is a superuser, so the administrator should either create a new superuser account and delete the cassandra account, or change the password for the cassandra account. Database administrators can then create users and Cassandra will manage user access to any of the nodes through users authenticating with their userid and password [22]. Administrators can also grant privileges to users to limit the access that a user has within the database [22]. Newly created non-superusers only have read access to a few tables in the “system” keyspace [22]. There is nothing in the Cassandra documentation about the capability or requirement for nodes to authenticate each other during internode communication.

b. Encryption

By default, there is no encryption of client-to-server communication or internode communication [22]. The database administrator can enable both of these through the configuration file to require SSL encryption for both. There do not appear to be any built-in functions in Cassandra for encrypting data for storage, so a user who wants data encrypted at rest in Cassandra must encrypt this data with another application before inserting the data into Cassandra.

c. Other

OpsCenter is a web based tool for accessing diagnostic and administrative information relating to the Cassandra database [26]. OpsCenter by defaults binds to all available network connections (0.0.0.0) on port 8888 [26]. By default, there is no authentication associated with the OpsCenter, but this can be enabled [26]. Using the default settings will allow anyone to connect to the OpsCenter who is aware of the IP address of the machine upon which OpsCenter is running.

5. Query Language

Communication with the Cassandra database is accomplished through CQL [22]. Users can employ either the CQL shell or another client program using the appropriate drivers. CQL is similar syntactically to SQL as both support the notion of a table consisting of columns and rows [25]. Creating keyspaces and tables in CQL is nearly identical to creating databases and tables in SQL [25]. This subsection examines the key differences between CQL and SQL.

One of the main differences is that CQL does not allow joins or subqueries since Cassandra is not a relational database [22]. Therefore, in the SELECT statement only one table is allowed in the FROM clause. Another difference revolves around the use of the WHERE clause. In SQL, a client can use any of the columns in a table for selecting rows from the table. In CQL, a client can only use columns that are part of the primary key for selecting rows from the table [25]. Even when the requested columns are part of the primary key, there are still restrictions on what columns can be used in the WHERE clause [25]. Consider the primary key {id, name, password}. A client will not be able to use only the column “password” in the WHERE clause. The client must also include at least the column “name” in the WHERE clause. Since data is stored on potentially many nodes and in sorted order, this restriction is necessary for efficient query operations. Therefore, it is incumbent on the database designer to wisely consider what queries are necessary when creating a table.

III. VULNERABILITY OVERVIEW AND METHODOLOGY

This chapter first briefly introduces the database vulnerabilities explored in this thesis: injection, misconfigured databases, HTTP interface, encryption, authentication and authorization. Then we explain the methodology used to examine each database management system (MySQL, MongoDB, and Cassandra) against each vulnerability. We conclude with a table summarizing the vulnerabilities in each database management system and introduce the remaining chapters that will examine each vulnerability more in depth and offer mitigations for these vulnerabilities.

A. OVERVIEW OF DATABASE VULNERABILITIES

This section introduces the vulnerabilities that we will later examine in depth in this thesis.

1. Injection

Injection is number one on the 2013 OWASP Top 10 list, which ranks the most critical security risks in web applications [7]. The 2011 CWE/SANS Top 25 list contains two vulnerabilities related to injection: “improper neutralization of special elements used in a SQL command” at number one and “reliance on untrusted inputs in a security decision” at number 10 [8]. Injection includes both SQL injection and NoSQL injection. Injection occurs when a user provides input for a SQL or NoSQL query that changes the intended meaning of the query [27]. The root cause of injection attacks is a failure to properly validate user input [27].

2. Misconfigured Databases

Security misconfiguration ranks at number five on the 2013 OWASP Top 10 list [7]. This vulnerability refers to the failure to change the default settings of a database management system (default accounts, network connection settings, and security settings) and the failure to properly apply patches [17].

3. HTTP Interface

This vulnerability refers to any interface that the database management system offers that allows someone to glean any data concerning the database. This could include administrative information, schema definitions, and the actual stored data.

4. Encryption

Sensitive data exposure, which can result from lack of encryption, ranks at number six on the 2013 OWASP Top 10 list [7]. "Missing encryption of sensitive data" ranks at number eight on the 2011 CWE/SANS Top 25 list [8]. This vulnerability includes both the encryption of data transmitting between two entities (e.g., client/server, node/node, etc.) and the encryption of data that is stored in the database [28].

5. Authentication and Authorization

"Missing authentication for critical function" and "missing authorization" rank numbers five and six respectively on the 2011 CWE/SANS Top 25 list [8]. These two vulnerabilities are related, but distinct from one another with authentication concerning itself with verifying a user's identity and authorization concerning itself with verifying a user's privileges or granting privileges to a user [21].

B. METHODOLOGY

We conducted our tests on one machine, an ASUS laptop running Microsoft Windows 8.1. The version of MySQL that we examined is MySQL 5.6 and MySQL Workbench 6.2. MongoDB's client and server is all contained in MongoDB 3.0.2. For Cassandra we used Datastax Community Edition 2.1.9 which includes Cassandra 2.1.6, CQL 3.2.0, and OpsCenter 5.1.3. We used Eclipse IDE Kepler Service Release 2 in order to build the front ends to connect to the three different database management systems.

We designed a simple entity-relationship diagram involving employees working in department at a company and then translated that model into schemas for each database management system. The diagram is depicted in Figure 2. Since this is an entity-relationship diagram, it translates very well to a MySQL schema. In MySQL the employee attribute “dno” is a foreign key reference to the department primary key “dnumber.” We use denormalized models to translate the Figure 2 diagram to MongoDB and Cassandra schemas with the employee attribute “dno” pointing to the department attribute “dnumber.” After building the tables, collections, and tables in MySQL, MongoDB, and Cassandra respectively, we populated each database with the same data.

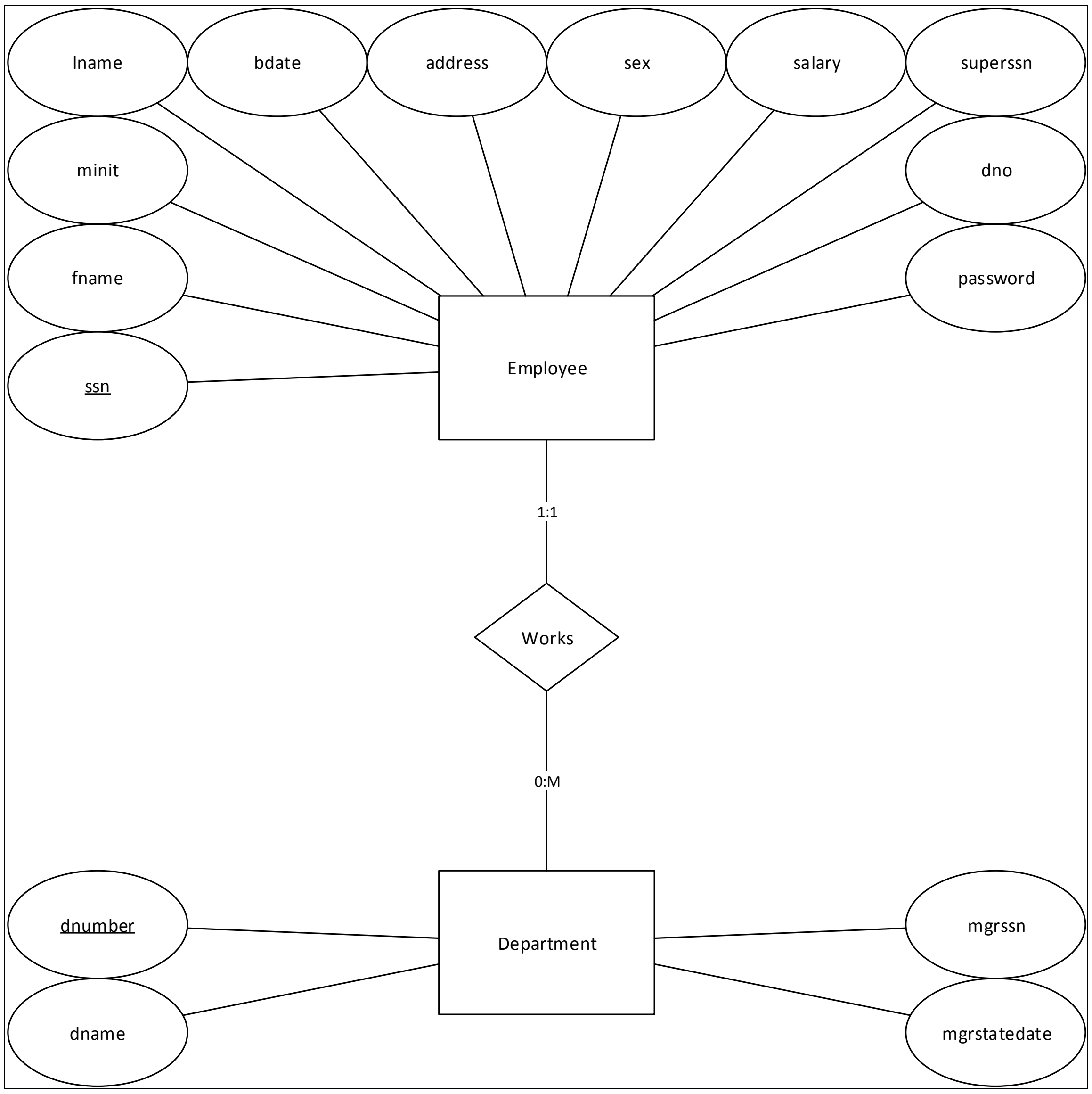


Figure 2. Database entity-relationship diagram.

A complete database system is normally made up of multiple tiers. The tiers are the data tier, the presentation tier, and an optional middle tier in between the data tier and the presentation tier [29]. The data tier is the database management system, and the presentation tier, also known as the front end, is what the user interfaces with to interact with the database management system [29]. We used a two tier model for our database system and built two different front ends for each database management system using the programming languages Python and Java. We wrote the programs using Eclipse and connected to each database management system using Python and Java drivers provided by MySQL, MongoDB, and Datastax (for Cassandra).

We then examined each database management system against the database vulnerabilities introduced in section IIIA. For the injection vulnerability we tested different injection techniques against different front end coding styles. To examine the misconfigured databases vulnerability we looked at the behavior of the database management system with default settings and then observed the behavior after changing settings. We looked at any HTTP interface offered by the database management system and assessed any security issues resulting from the interface. For the encryption, authentication, and authorization vulnerabilities, we examined the default behavior of each database management system and looked at what capabilities each system offers regarding each area. The chapters on each vulnerability cover our approach in more detail.

Table 1 summarizes the database vulnerabilities for each database management system. Chapters IV through VIII cover in more detail how we examined each database management system against the vulnerability, discuss our findings, and present mitigations against the vulnerability. Chapters IV, V, VI, VII, and VIII cover injection, misconfigured databases, HTTP interface, encryption, and authentication and authorization respectively.

Table 1. Database vulnerabilities. Presentation tiers in Python and Java.

Problem	MySQL	MongoDB	Cassandra
1. Injection			
a. Tautologies	Vulnerable	Vulnerable	Not Vulnerable
b. Illegal query	Vulnerable	Vulnerable	Vulnerable
c. Union query	Vulnerable	Not Vulnerable	Not Vulnerable
d. Piggy-back query	Vulnerable, but not by default	Vulnerable	Not Vulnerable
2. Misconfigured databases	- Use of default port - Bind to 0.0.0.0 by default	- Use of default port - Bind to 0.0.0.0 by default	- Use of default port - Bind to localhost by default - Default keystore password
3. HTTP Interface	None	- Yes, disabled by default - Use of default port	- Yes, disabled by default - Use of default port - Bind to 0.0.0.0 by default
4. No encryption	- No client/server encryption by default - Built-in functions exist for encrypting data	- No client/server encryption by default - No inter-node encryption by default - No built-in functions exist for encrypting data	- No client/server encryption by default - No inter-node encryption by default - No built-in functions exist for encrypting data
5. No authentication	Enabled by default	Disabled by default	Disabled by default
6. No authorization	Enabled by default	Disabled by default	Disabled by default

THIS PAGE INTENTIONALLY LEFT BLANK

IV. INJECTION

This chapter examines the injection vulnerability. We first look at the severity and scope of the vulnerability. We then provide a definition of injection, which includes examining the source of injection attacks, the intent of the attacker, and categories of injection. Next we examine prevention methods suggested in literature and then conclude with our results from testing MySQL, MongoDB, and Cassandra and provide mitigations to combat the injection vulnerability in each DMBS where applicable.

A. SEVERITY AND SCOPE OF INJECTION VULNERABILITY

The SQL injection vulnerability was first discovered in 1998 [30]. Since then, researchers have written multitudes of papers on both SQL and NoSQL injection problems. Despite the massive attention devoted to understanding and preventing the injection vulnerability since 1998, it still takes the top spot in the 2013 OWASP Top 10 Most Critical Web Application Security Risks [7].

Consequences are dire for organizations and companies that have vulnerable web applications. An attacker can retrieve information from and write information to the underlying database in ways not intended by the application designer. An attacker can also take control of the database, destroy the database, and even gain access to the computer on which the database is installed [27].

While the SQL injection vulnerability is well known, it is important to note that NoSQL DBMS are also vulnerable to injection despite not using the SQL language. NDDBMS such as MongoDB and CouchDB both use a JSON based query language, but both of these are vulnerable to injection [31].

B. DEFINITION OF INJECTION

The injection vulnerability refers to both SQL and NoSQL injection. Both of these injection types occur when a user provides input through a presentation tier and, that input is included in the subsequent query to the data tier so that the input is considered as query language code [27]. For example, a SQL query template in the presentation tier may be

“select * from employee where password = ‘“ + input + “‘“

where “input” is what the user will provide. This simple example allows an employee to view his or her own information. A non-malicious user may provide 12345 as input and the subsequent query to the data tier would be “select * from employee where password = ‘12345.’” However, a malicious user may provide

‘ or 1=1

as input, and the subsequent query to the data tier would be “select * from employee where password = ‘‘ or 1=1.” The query is now fundamentally different from what was intended by the presentation tier query template as the input is treated as query language code.

We define injection vulnerability as a condition that exists where a user is able to provide input so that when combined with the presentation tier query template, the resulting query structure is syntactically different from what was intended by the query template designer [27]. In order to better understand the injection vulnerability, we also examine it in terms of injection attack sources, injection intention, and categories of injection attacks.

1. Injection Attack Sources

The input in an injection attack can come from a variety of sources. The most well understood source is that of a user providing some form of input through a presentation tier [27]. An example would be a web application that allows a user to provide input in a text box. Another source is from a user cookie. websites use the data in cookies to make decisions. An attacker could modify the data in a cookie to exploit an injection vulnerability [27]. A third source is server variables [27]. When a user makes a HTTP request using the GET method, the user can specify several optional fields in the request. These fields are treated as variables that the web server uses to answer a HTTP request. A malicious user could craft the HTTP request to target an injection vulnerability. Other possible sources include anything that is physically inputted into a database such as scanned documents, barcodes, and RFID tags [32].

2. Injection Intention

An attacker may have multiple goals when attacking a database server and so each injection attempt will probably have a specific intention. In the early stages of an attack the intent of an injection will be to determine if there is indeed a vulnerability and where the vulnerability occurs [27]. Another injection intention in this early stage will be to determine what kind of DBMS the server is running [27]. As the attack progresses, another injection intention may be to determine information regarding the schema of the database (i.e., tables and columns in MySQL and collections and documents in MongoDB) [27]. As the attacker gathers additional information about the DBMS, subsequent intentions of injections may be to extract data from the database, load data to the database, circumvent authentication and authorization protections, conduct denial of service by dropping the database or specific tables/collections, or attempt to execute commands on the underlying system on which the DBMS is running [27].

3. Categories of Injections

There are four major categories of injections that we consider: tautologies, illegal or logically incorrect queries, union queries, and piggy-back queries.

a. Tautologies

The first category of injection is tautology. A tautology is an expression that is always true. Instinctively, the goal of a tautology injection is to target the conditional portion of a query so that the condition is always true upon evaluation [27]. The example that we provided in Section B of this chapter is a form of tautology injection. An attacker will normally use a tautology injection in order to circumvent an authentication mechanism with a further goal of obtaining data [27].

b. *Illegal or Logically Incorrect Queries*

The second category of injection is illegal or logically incorrect queries. The goal of this type of injection is to collect information about the database or the DBMS [27]. The proximate cause of this vulnerability is that the DBMS provides too much useful information in the resulting error message [27]. Such useful information would be database schema or type and version of DBMS. Using the presentation tier query template in Section B, an example of this injection would be

```
‘ or cast(“hello world” as INT) #
```

In MySQL “#” is used as a comment marker. This intent of including this is so that MySQL will ignore any characters that may occur after the “#” character. “INT” is not a valid argument for the “cast” function and will result in an error message that reveals the DBMS is MySQL: “You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version.”

c. *Union Queries*

The third category of injection is union queries. The goal of this type of injection is to append an additional query onto the original request to retrieve additional information from that database [27]. In order to effectively use a union query, the attacker requires knowledge of the database schema. If an attacker is able to determine the DBMS type (perhaps through an illegal or logically incorrect query), then the attacker could leverage knowledge of the administrative databases within a DBMS and use a union query to determine the names of other databases, tables, and columns within the DBMS. Using the presentation tier query template in Section B, an example of this injection would be

```
‘ union select mgrssn, dname from department #
```

This will allow the attacker to ascertain all the department names and the social security number of each department manager.

d. Piggy-back Queries

The final category of injection is piggy-back queries. The goal of this type of injection is to insert multiple queries into the database [27]. The piggy-back query takes advantage of DBMS that allow multiple queries using a special symbol (such as a semi-colon) to separate the queries [27]. An attack of this type can be used to achieve many goals. An attacker could retrieve additional information, add information to the database, conduct a denial of service attack, or even execute special stored procedures the DBMS may offer for accessing the underlying computer system [27]. Using the presentation tier query template in Section B, an example of this injection would be

‘; create database foo #

This will allow the attacker to create a new database in the DBMS called “foo.”

C. SUGGESTED PREVENTION METHODS IN LITERATURE

The existence of an injection vulnerability may not be the fault of the DBMS manufacturer, but rather the presentation tier designer. Such vulnerabilities may manifest in poor coding practices or failure to validate user input. However, presentation tiers must make use of software drivers for communicating with the data tier that are provided by the manufacturer. In some cases the drivers may offer methods for preventing the injection vulnerability.

Since there is no guarantee that the presentation tier, DBMS driver, or data tier will be error free, most prevention methods suggested in literature focus on proposing software that acts as an intermediary between the presentation tier and the data tier. The proposed software will then attempt to defeat attacks exploiting an injection vulnerability. This section examines these and other prevention methods.

1. Use Good Coding Practices

Most solutions proposed in the literature first suggest the use of good coding practices in the presentation tier [27]. One technique is using a strongly typed programming language [27]. For example, this allows the designer to specify that user input must be an integer. If a user provides input of type string, then there will be a syntax error. Even without using a strongly typed language, presentation-tier designers should check the user input to verify that the input appears to be an integer in the above example. Another technique is to encode user input so characters that could be used in the DBMS query language (such as apostrophes and semi-colons in SQL), are not treated as such when the DBMS processes the query [27]. A third technique would be to check that user input satisfies certain conditions [27]. For example, a designer could verify that input for a password field does not contain apostrophes or semi-colons and also does not exceed a specified length of characters.

While using good coding practices is effective, it is difficult to ensure across an organization that presentation tier designers will create error-free implementations of these techniques [33]. It can also be cost prohibitive to comb through legacy presentation tiers to correct programming errors or redesign them completely [33]. The solutions found in literature therefore propose an automated approach that acts as a go-between the presentation tier and the data tier.

2. Combine Static and Dynamic Analysis

Several solutions presented in literature suggest using a combination of static and dynamic analysis to prevent injection. One such suggested solution is called AMNESIA. This tool first performs static analysis of a Java-written presentation tier [34]. The result of the static analysis is both a list of points in the code where the presentation tier sends queries to the DBMS and models of legal queries for those points [34]. The tool performs dynamic analysis when the presentation tier actually sends a query to the DBMS [34]. It compares the user query with the model of legal queries generated during static analysis [34]. If the user query matches a legal query then the user query is passed to the DBMS, otherwise the user query is rejected [34].

Other authors propose a static and dynamic analysis tool using finite state machines [33]. Their tool performs static analysis of a presentation tier and generates a finite state machine representing legal queries for every point in the code where the presentation tier sends queries to the DBMS [33]. During dynamic analysis the presentation tier sends a query to the DBMS and the tool runs the query through the finite state machine for the associated point in the code where the finite state machine was constructed [33]. If the finite state machine accepts the user query, then the tool passes the query to the DBMS, otherwise the tool rejects the query [33]. The accuracy of the models and finite state machines in these two approaches determine how successful the tool is in avoiding false positives (rejecting a legal query) and false negatives (allowing an injection) [27].

3. Parse Trees

Another set of suggested solutions use parse trees to test for injection, one of which is a tool called SQLGuard. SQLGuard provides a method to wrap each SQL query in the presentation tier [29]. This method accepts user input using a generated key to ensure that neither the input, nor the query structure is modified [29]. Another method would verify and remove the key and then build two parse trees [29]. The first parse tree used the grammar of the SQL query prior to user input and the second parse tree included user input [29]. This verification would then compare the two parse trees to verify that the query had not changed syntactically [29]. One drawback of this method is that the presentation tier developer must modify each portion of code containing a query to introduce the method wrapper and the verifying method. Another potential drawback would be that if a malicious user can guess the key, then he or she could defeat SQLGuard.

4. Instruction Set Randomization

A fourth suggested solution is to randomize the DBMS instruction set [27]. A tool that uses this approach is called SQLrand. The authors use a secret key to generate keywords that replaces the standard SQL keywords (such as insert, select, drop, etc.) [27]. Any malicious user attempting to inject statements would use standard SQL keywords and so the resulting query would not be in the correct syntax for the DBMS. This tool could be defeated if the attacker could learn the secret key [27].

5. Tainting

Another set of suggested approaches involved the use of negative and positive tainting. Tainting refers to treating user input as potentially malicious [35]. Any information flow with user input attached would have a negative taint and otherwise would have positive taint [35]. One suggested tool that uses positive tainting is called MetaStrings. This tool identifies those strings that had positive taint [35]. Before a query was sent to the DBMS, the tool would check to see if the keywords had only positive taint. If not, the query was rejected, otherwise it was sent to the DBMS [35].

6. Replace String Concatenation with Parameterization

In the previous subsections there is an assumption that the presentation tier makes use of string concatenation to dynamically build queries with user input. The tools then seek to prevent a malicious query from reaching the DBMS. Since string concatenation is what malicious users normally exploit to create SQL injection, some authors propose creating new methods of generating queries and an application programming interface (API) that would allow the presentation tier to communicate with the DBMS using these new methods. One such tool is called SQL DOM. SQL DOM examines the database schema in a DBMS and generates a dynamically linked library (DLL) that is used in between the presentation tier and the DBMS [36]. The DLL uses strongly typed classes and input checking to dynamically generate queries that prevents injection. Presentation tier developers would have to write queries that conforms to the API that the DLL provides [36].

MySQL now provides methods for presentation tier developers to prevent injection called parameterized queries that are available in several languages to include Python and Java [16]. While SQL DOM and parameterized queries successfully prevent injection, the presentation tier developers are forced to conform to a new coding method. It can be a costly process for presentation tiers developers to change code in legacy systems to conform to new methods [27].

D. TESTING RESULTS AND MITIGATIONS

We tested each DBMS against each category of injection. This section examines the test results and provides mitigations where applicable.

As noted in Chapter III, we built two presentation tiers using Python and Java. The query template we used in Python and Java for MySQL and Cassandra is “select * from employee where password = ‘“ + user_input + .”‘“ The non-malicious user is expected to provide an input, for example, “aaa,” and the query would return the employee whose password is equal to “aaa.”

The query template we used in Python and Java for MongoDB is “db.employee.find({“password”:user_input}).” In Python 3.X user_input is accepted as “eval(input(“provide input: “)).” In Java user_input is accepted using the “scanner” utility.

1. Tautologies

This subsection presents the test results of tautology injection against MySQL, MongoDB, and Cassandra. Mitigations are also presented where applicable.

a. MySQL

A tautology example for MySQL would be injecting “‘ or 1=1 #” resulting in the query “select * from employee where password = ‘‘ or 1=1 #.” This query evaluates each row, and even though the password is null and does not match, the “or 1=1” causes the entire condition to evaluate to true for every row. Therefore, all employee information will be returned. This example bypasses the authentication we have in place and extracts data that the user is not permitted to see. Both the Python and Java presentation tiers are vulnerable to this injection.

The presentation tier developer can mitigate this vulnerability by using the parameterized queries provided in both Python and Java. In Python the new query template would be queryTemplate = “select * from employee where password = %(password)s,” and the parameterized input is set with “cur.execute(queryTemplate, {‘password’ : user_input}).” In Java the new query template would be queryTemplate =

con.preprareStatement(“select * from employee where password = ?”), and the parameterized input is set with “queryTemplate.setString(1, user_input).”

b. MongoDB

A tautology example for the Python presentation tier to MongoDB would be injecting {“$ne”:”1”} resulting in db.employee.find({“password”:{“$ne”:”1”}}). This query evaluates every document and returns every password that is not equal to “1.” This is not a pure tautology, but behaves like one. Java is not vulnerable to this type of attack as the variable user_input must be declared as a specific data type supported by Java. Java supports byte, short, int, long, float, double, Boolean, char, and string datatypes [36]. Since {“$ne”:”1”} is an object, such user input will cause a Java syntax error.

The presentation tier developer can mitigate the Python vulnerability by avoiding the use of “eval()” in reading user input in Python 3.X and “input()” in Python 2.X. The developer should instead use “input()” in Python 3.X and “raw_input()” in Python 2.X.

c. Cassandra

Cassandra is not vulnerable to this category of injection as the Python and Java drivers for Cassandra detect a syntax error when attempting to parse a comment marker (“--” for CQL).

2. Illegal or Logically Incorrect Queries

This subsection presents the testing results of illegal or logically incorrect query injection against MySQL, MongoDB, and Cassandra and mitigations where applicable.

a. MySQL

An example for MySQL would be injecting “‘ and 1 = (select * from projects) #” resulting in the query “select * from employee where password = ‘‘ and 1 = (select * from projects) #.” This will return the error “Table ‘cs3060db1.projects’ doesn’t exist.” The attacker could repeat this many times in order to guess what tables may exist in the database. Also note this error reveals the name of the database in use. This vulnerability exists in both the Python and Java presentation tiers. The presentation tier developer can mitigate this vulnerability by using parameterized queries as discussed in Section 1a.

b. MongoDB

MongoDB does not allow subqueries, therefore it does not have the level of vulnerability that MySQL has regarding this category of injection. However, the presentation tier driver may reveal useful information in error messages. An example for the Python presentation tier would be injecting "db.employee.find()" resulting in the query db.employee.find({"password": db.department.find()}). This will return the error "bson.errors.InvalidDocument: Cannot encode object: <pymongo.cursor.Cursor object at 0x*address*>." A malicious user could key in on the word "pymongo" to deduce that the associated DBMS is MongoDB. Our presentation tier written in Java did not reveal any information at all. The Python presentation tier developer can mitigate this vulnerability by encapsulating outputs to the user in a try/except block. This will ensure the error messages do not reach the user.

c. Cassandra

Cassandra does not allow subqueries, therefore it also does not yield the level of vulnerability that MySQL has regarding illegal and illogical queries. However, both the Python and Java presentation tiers yield useful information in their syntax error messages. Using the same example injection offered in Section 2a, Python returns the error message "cassandra.protocol.SyntaxException: <ErrorMessage code=2000 [Syntax error in CQL query] message="line 1:115 mismatched character '<EOF>'expecting '''">." The word "cassandra" in this error message will inform a malicious user that the DBMS is Cassandra. Java returns the error message "Exception in thread "main" com.datastax.driver.core.exceptions.SyntaxError: line 1:115 mismatched character '<EOF>' expecting ''.'" The word "datastax" will inform a malicious user that the DBMS is Cassandra. The Python presentation tier developer can mitigate this vulnerability using the technique discussed in Section 2b. The Java presentation tier developer can mitigate this vulnerability by redirecting error messages to a file using the code "File file = new File("error.txt"); System.setErr(new PrintStream("error.txt"));."

3. Union Queries

This subsection presents the test results of union query injection against MySQL, MongoDB, and Cassandra. Mitigations are presented where applicable.

a. MySQL

A malicious user would first submit a legitimate query to determine how many columns are returned to the user since the number of columns returned in the select statement in the union query must match the number of columns in the first select statement. If a legitimate query returns two columns, then an example of injection would be "' union select schema_name, catalog_name from information_schema.schemata #." This will return all the database names in the MySQL server. The column "catalog_name" is used only to ensure that two columns are returned from the injected select statement.

Once the malicious user has the database names, then he or she can inject the input "'union select table_name, column_name from information_schema.columns where table_schema = "cs3060db1" #" to determine the tables in the database and each table's column names. Now the malicious user can inject the input "'union select mgrssn, dname from department #" to learn the SSNs of the managers in the company and their department names.

This vulnerability exists in both the Python and Java presentation tiers. This can be mitigated using the parameterized query technique shown in Section 1a. Another mitigation for this example involves controlling access to the "information_schema" database. The database administrator should ensure that the user account that MySQL is running under does not have access to the "information_schema" database.

b. MongoDB and Cassandra

Neither MongoDB nor Cassandra are vulnerable to this category of injection because neither DBMS' query language supports union queries.

4. Piggy-back Queries

This subsection presents the testing results of piggy-back query injection against MySQL, MongoDB, and Cassandra and mitigations where applicable.

a. MySQL

In order for MySQL to be vulnerable to this type of injection, the presentation tier must allow for multiple queries to be executed in one statement. By default Python and Java do not allow this, but do offer support for changing the default value. If the presentation tier developer changes the default value to allow multiple queries, then MySQL will be vulnerable. A malicious user could inject the input "' ; create table foo (foo1 int, primary key (foo1)) #" that would result in the creation of a new table in the active database called "foo." There are two mitigations of this vulnerability. The first is to not change the default setting for multiple queries in the Python and Java presentation tiers. The second is to use parameterized queries.

b. MongoDB

This example demonstrates how a malicious user can use an injection to cause MongoDB to execute an unintended query on his or her behalf and return the result of the query to the malicious user. The user can inject "exec("print(db.command(\"listCollections\"))")" resulting in the query "db.employee.find("password": exec("print(db.command(\"listCollections\"))"))." This will print out all the collections in the current database. The Python presentation tier is vulnerable to this injection when "eval()" is used in Python 3.X or "input()" is used in Python 2.X for accepting user input. The mitigation for this vulnerability is to instead use "input()" in Python 3.X or "raw_input()" in Python 2.X for accepting user input. The Java presentation tier is not vulnerable to this injection as the equivalent of "eval()" is not available in standard Java syntax.

c. Cassandra

Cassandra is not vulnerable to this category of injection as the Python and Java drivers for Cassandra detect a syntax error when attempting to parse a comment marker ("--" for CQL).

V. MISCONFIGURED DATABASES

This chapter examines the vulnerabilities arising from misconfigured databases. We first provide a definition of misconfigured databases. We finish with examining MySQL, MongoDB, and Cassandra for any misconfigured database vulnerabilities and provide mitigations to remedy these vulnerabilities.

A. DEFINITION OF MISCONFIGURED DATABASES

Misconfigured databases can be a major problem for an organization. The 2013 OWASP Top 10 Most Critical Web Application Security Risks ranked "Security Misconfiguration" at number five on the list [7]. We define a misconfigured database as a DBMS in which the database administrator has not modified the default configuration settings, adjusted default settings arising from choosing an installation option, or applied security patches from the manufacturer [17]. The DBMS may very well be unsecured when used after just the standard installation with no further adjustments or modifications. The default configuration settings may allow any individual with an Internet connection to interact with the DBMS. Additional features provided by the DBMS manufacturer may allow an attacker to take control of the underlying system on which the DBMS is installed [17]. When a manufacturer releases security patches, both clients of the software and malicious users learn of the patches. An attacker can deduce the DBMS vulnerabilities and create attacks that can exploit the unpatched software.

The default configurations settings that we consider in this chapter are network settings that may leave the DBMS open to the Internet. There are additional default configuration settings that we examine in the following chapters on encryption, authentication, and authorization.

B. VULNERABILITIES AND MITIGATIONS

This section examines misconfigured database vulnerabilities in MySQL, MongoDB, and Cassandra and presents mitigations against those vulnerabilities where applicable.

1. MySQL

The first vulnerability we examine is default network configuration settings. During installation of MySQL, an option is to enable TCP/IP networking [16]. If this is not enabled, then only localhost connections are possible. This portion of the installation also allows the administrator to specify the port number that MySQL will use and specify whether or not to open the firewall for this port. The default options for this section of installation is to enable TCP/IP networking, specify 3306 as the port, and to open the firewall. The network bind address default is 0.0.0.0 [16]. This means that MySQL will attempt to bind to all network interfaces on the underlying system. The end result of these configurations settings is that anyone who is able to access the system on which MySQL is installed (i.e., ping the IP address of that system) will also be able to interact with MySQL on port 3306.

There are a few mitigations to consider for this vulnerability. One mitigation is to change the default port number from 3306 to another port. If an attacker is specifically scanning the Internet for computers open on port 3306 for the express purpose of attacking MySQL installations, this will at least hide the fact that the computer is running MySQL. This of course will also require any presentation tier developers to modify their code so that the presentation tier will connect on the new specified port. If it is paramount that MySQL be available to other systems on the network, then another mitigation is to only allow connections through the system firewall from specific IP addresses. A third mitigation is to only allow MySQL to bind to localhost so that no outside connections to MySQL are permitted. In order for other computers on the network to make use of MySQL in this instance, the server accepting requests from the presentation tier must exist on the same system on which MySQL is installed. Users would connect to the presentation tier and then the presentation tier will make a localhost connection to MySQL.

Another potential vulnerability is that default error messages returned by MySQL can reveal to a user that the DBMS in use is MySQL. This information allows an attacker to craft exploits specific to MySQL. Error messages can also reveal to an attacker information about the database (e.g., database does not exist, table does not exist).

MySQL offers a mitigation for this vulnerability by providing a mechanism to the database administrator to change the content of error messages [16].

2. MongoDB

MongoDB's default networking configuration is vulnerable to remote discovery and reconnaissance. The process of installing MongoDB is a very simple one with no options to choose from. By default MongoDB binds to all networking interfaces (0.0.0.0) on port 27017 [21]. If a database administrator installs MongoDB on a system that is directly connected to the Internet, anyone will be able to see that this system is open on port 27017. Shodan is an Internet of Things search engine that maintains a database of devices connected to the Internet. Heyens et al. conducted an experiment where they queried Shodan for port 27017 and found nearly 40,000 devices open on port 27017 [38].

The mitigations for this vulnerability are similar to the mitigations listed in section B.1. Database administrators can change both the port and bind address in the MongoDB configuration file.

3. Cassandra

The default networking configuration is not a vulnerability for Cassandra. Cassandra does use a default port for CQL clients (9042), but the default bind address is localhost [22]. A database administrator who installs Cassandra on a system that is directly connected to the Internet will not have to worry about the system being open on port 9042 using the default configuration settings.

Cassandra does, however, have a configuration vulnerability related to server encryption. When using encryption, the Java keystore is used to store the private key that Cassandra utilizes for encrypting outgoing messages. The Truststore is used to store the trusted certificate that Cassandra utilizes to authenticate remove servers. The default password for both the Java Keystore and the Truststore is "cassandra." [22] The mitigation for this vulnerability is changing the default password, which can be specified in the Cassandra configuration file.

THIS PAGE INTENTIONALLY LEFT BLANK

VI. HTTP INTERFACE

This chapter examines the vulnerabilities arising from HTTP Interfaces. We first provide a definition of HTTP Interfaces. We finish with examining MySQL, MongoDB, and Cassandra for any HTTP Interface vulnerabilities and provide mitigations to remedy these vulnerabilities.

A. DEFINITION OF HTTP INTERFACE

HTTP Interfaces are not mentioned specifically in the 2013 OWASP Top 10 Most Critical Web Application Security Risks list or the 2011 CWE/SANS Top 25 list, but these two documents enumerate many risks that this vulnerability falls under: “Sensitive Data Exposure,” “Cross-Site Request Forgery,” “Missing Authorization,” and “Missing Authentication for a Critical Function” [7][8].

We define HTTP Interface as any service or API that the DBMS offers in order to interact with the DBMS in some way. This can be something as simple as pointing a web browser to the IP address of the system on which the DBMS is installed and the port on which the DBMS offers the service. This type of service may offer configuration information about the database, the DBMS, or the system on which the DBMS is installed. This may be displayed upon the landing page or accessible through links using prebuilt DBMS commands. Someone with an understanding of DBMS commands and usage through the HTTP Interface may be able to execute other commands through the DBMS besides the prebuilt commands. The DBMS may also offer an API that a developer can use to build an application like a presentation tier to interact with the DBMS. Not all DBMS that offer HTTP Interfaces include support for authentication even though authentication may be set for traditional interaction with the DBMS [21]. Even if the DBMS in on a system within a secure network, an attacker may be able to access the DBMS through cross-site request forgeries [39].

B. VULNERABILITIES AND MITIGATIONS

This sections examines HTTP Interface vulnerabilities in MySQL, MongoDB, and Cassandra and presents mitigations against those vulnerabilities where applicable.

1. MySQL

The version of MySQL that we tested is not vulnerable to this as it does not include a HTTP Interface service or API.

2. MongoDB

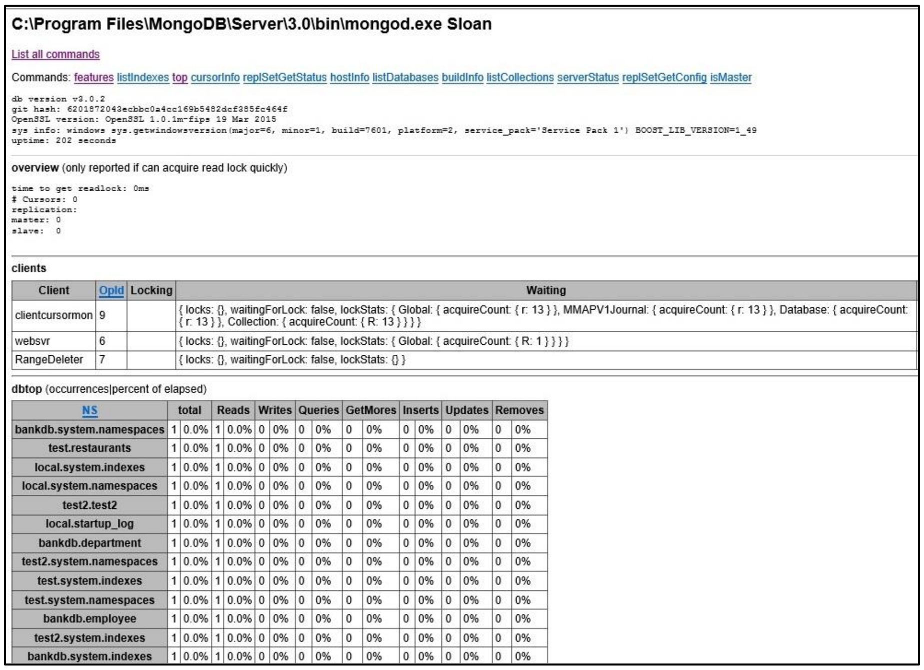
MongoDB supports a HTTP interface that provides access to the databases. By default this feature is not enabled in MongoDB [21]. The database administrator can enable the HTTP interface in the MongoDB configuration file [21]. The interface is available at IP:PORT where IP is the IP address of the system and PORT is 1000 more that the MongoDB port (if the MongoDB port is 27017, then the HTTP interface port is 28017) [21].

When enabled, the interface provides a non-interactive display of administrative information about the databases and the system on which MongoDB is installed [21]. Figure 3 shows an example of information available through the interface. Note that this interface provides complete file path, OpenSSL version, the database version, information about the system, and names of both databases and collections. There are also some links displayed to built-in commands, but these links do not work unless the HTTP interface is in full interactive mode [21].

The database administrator can make the HTTP interface fully interactive by also enabling this option in the MongoDB configuration file [21]. When enabled, the links shown in Figure 3 are active, and a user can also submit queries to the database. Since the names of the databases and collections are shown on the HTTP interface landing page, it is easy for a user to display all the documents in a collection with the URL

http://IP:PORT/db_name/collection_name/

Figure 4 shows all the documents displayed in the "employee" collection using this technique.



C:\Program Files\MongoDB\Server\3.0\bin\mongod.exe Sloan

List all commands

Commands: features listIndexes top cursorInfo replSetGetStatus hostInfo listDatabases buildInfo listCollections serverStatus replSetGetConfig isMaster

```
db version v3.0.2
git hash: 6201872043ecbbc0a4cc169b5482dcf385fc464f
OpenSSL version: OpenSSL 1.0.1m-fips 19 Mar 2015
sys info: windows sys.getwindowsversion(major=6, minor=1, build=7601, platform=2, service_pack='Service Pack 1') BOOST_LIB_VERSION=1_49
uptime: 202 seconds
```

overview (only reported if can acquire read lock quickly)

```
time to get readlock: 0ms
# Cursors: 0
replication:
master: 0
slave:  0
```

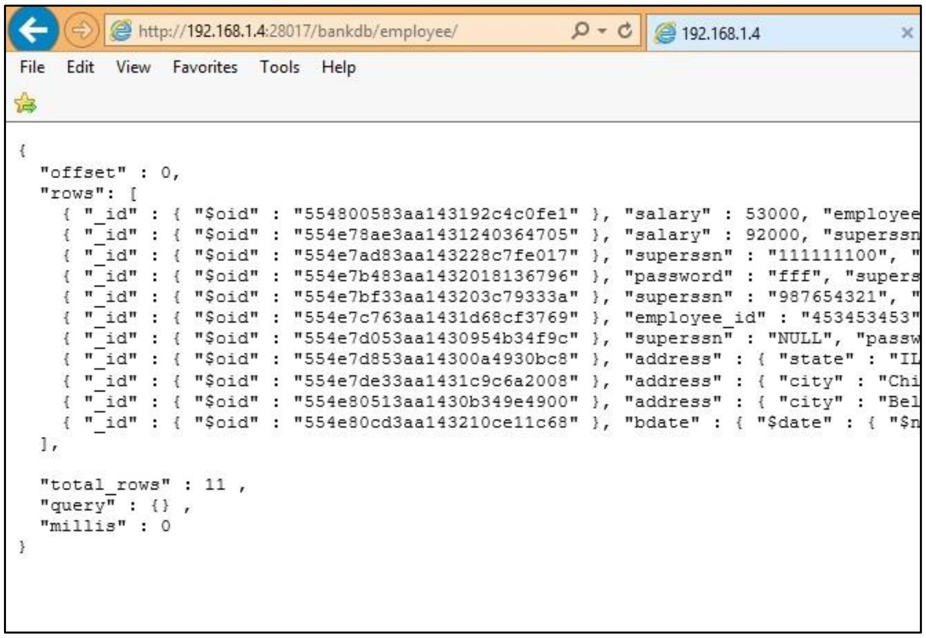
clients

Client	OpId	Locking	Waiting
clientcursormon	9		{ locks: {}, waitingForLock: false, lockStats: { Global: { acquireCount: { r: 13 } }, MMAPV1Journal: { acquireCount: { r: 13 } }, Database: { acquireCount: { r: 13 } }, Collection: { acquireCount: { R: 13 } } } }
websvr	6		{ locks: {}, waitingForLock: false, lockStats: { Global: { acquireCount: { R: 1 } } } }
RangeDeleter	7		{ locks: {}, waitingForLock: false, lockStats: {} }

dbtop (occurrences\|percent of elapsed)

NS	total		Reads		Writes		Queries		GetMores		Inserts		Updates		Removes	
bankdb.system.namespaces	1	0.0%	1	0.0%	0	0%	0	0%	0	0%	0	0%	0	0%	0	0%
test.restaurants	1	0.0%	1	0.0%	0	0%	0	0%	0	0%	0	0%	0	0%	0	0%
local.system.indexes	1	0.0%	1	0.0%	0	0%	0	0%	0	0%	0	0%	0	0%	0	0%
local.system.namespaces	1	0.0%	1	0.0%	0	0%	0	0%	0	0%	0	0%	0	0%	0	0%
test2.test2	1	0.0%	1	0.0%	0	0%	0	0%	0	0%	0	0%	0	0%	0	0%
local.startup_log	1	0.0%	1	0.0%	0	0%	0	0%	0	0%	0	0%	0	0%	0	0%
bankdb.department	1	0.0%	1	0.0%	0	0%	0	0%	0	0%	0	0%	0	0%	0	0%
test2.system.namespaces	1	0.0%	1	0.0%	0	0%	0	0%	0	0%	0	0%	0	0%	0	0%
test.system.indexes	1	0.0%	1	0.0%	0	0%	0	0%	0	0%	0	0%	0	0%	0	0%
test.system.namespaces	1	0.0%	1	0.0%	0	0%	0	0%	0	0%	0	0%	0	0%	0	0%
bankdb.employee	1	0.0%	1	0.0%	0	0%	0	0%	0	0%	0	0%	0	0%	0	0%
test2.system.indexes	1	0.0%	1	0.0%	0	0%	0	0%	0	0%	0	0%	0	0%	0	0%
bankdb.system.indexes	1	0.0%	1	0.0%	0	0%	0	0%	0	0%	0	0%	0	0%	0	0%

Figure 3. Information available through MongoDB HTTP Interface.


http://192.168.1.4:28017/bankdb/employee/
192.168.1.4
File Edit View Favorites Tools Help

```
{
  "offset" : 0,
  "rows": [
    { "_id" : { "$oid" : "554800583aa143192c4c0fe1" }, "salary" : 53000, "employee
    { "_id" : { "$oid" : "554e78ae3aa1431240364705" }, "salary" : 92000, "superssn
    { "_id" : { "$oid" : "554e7ad83aa143228c7fe017" }, "superssn" : "111111100", "
    { "_id" : { "$oid" : "554e7b483aa1432018136796" }, "password" : "fff", "supers
    { "_id" : { "$oid" : "554e7bf33aa143203c79333a" }, "superssn" : "987654321", "
    { "_id" : { "$oid" : "554e7c763aa1431d68cf3769" }, "employee_id" : "453453453"
    { "_id" : { "$oid" : "554e7d053aa1430954b34f9c" }, "superssn" : "NULL", "passw
    { "_id" : { "$oid" : "554e7d853aa14300a4930bc8" }, "address" : { "state" : "IL
    { "_id" : { "$oid" : "554e7de33aa1431c9c6a2008" }, "address" : { "city" : "Chi
    { "_id" : { "$oid" : "554e80513aa1430b349e4900" }, "address" : { "city" : "Bel
    { "_id" : { "$oid" : "554e80cd3aa143210ce11c68" }, "bdate" : { "$date" : { "$n
  ],

  "total_rows" : 11 ,
  "query" : {} ,
  "millis" : 0
}
```

Figure 4. Employee collection displayed through MongoDB HTTP Interface.

The MongoDB HTTP interface does not support authentication, even if authentication is enabled for normal MongoDB transactions [21]. If the HTTP interface and interactive mode are enabled with the default network configuration, then anyone who is able to reach the machine on which MongoDB is installed will be able to query the database.

There are two mitigations for this vulnerability. The first is to not enable the HTTP interface unless it is absolutely necessary. Under no circumstances should the database administrator enable full interactive mode if using the default network configuration. The second mitigation is to change the network configuration to only allow connections from the localhost. In this case, enabling this HTTP interface and full interactive mode will not present a risk to the organization unless an attacker is able to take control of the machine remotely.

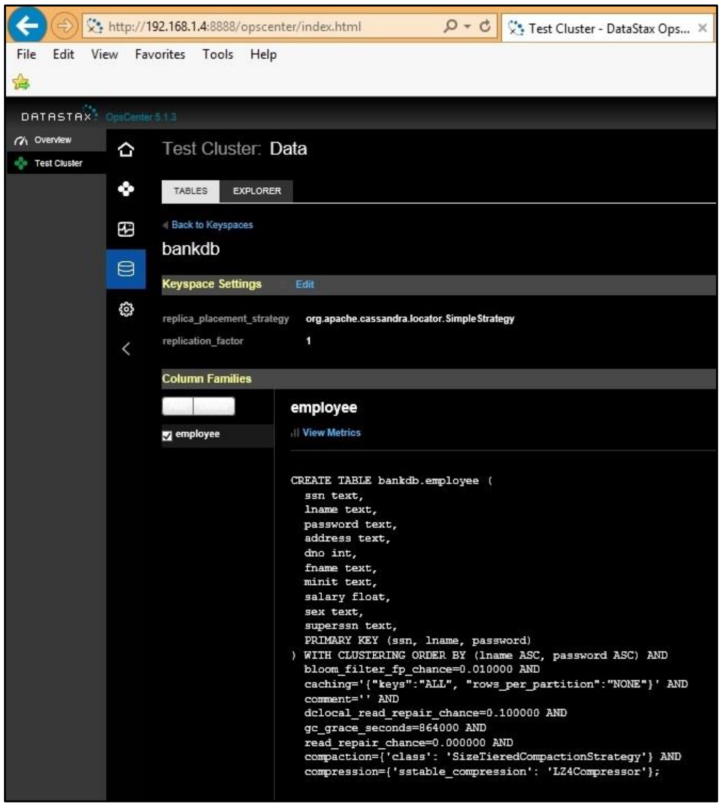
3. Cassandra

Datastax Community Edition provides OpsCenter as an installation option along with Cassandra. OpsCenter is a web based tool for accessing diagnostic and administrative information relating to the Cassandra database [26]. The configuration file for OpsCenter is separate from the configuration file for Cassandra. By default the OpsCenter bind address is all network interfaces (0.0.0.0) on port 8888 [26]. This differs from Cassandra's default bind address, which is 127.0.0.1 (localhost) [25]. By default there is no authentication or encryption for OpsCenter, but this can be enabled in the configuration file [26].

OpsCenter provides both information and utilities for the database administrator to interact with the cluster and keyspaces within the cluster. An individual accessing OpsCenter can see the version of Cassandra, a list of all the nodes in the cluster along with their IP addresses, a list of the keyspaces stored in the cluster, a list of the tables stored in each keyspace, and the CQL used to define each table. Figure 5 shows an example of information about the "employee" table available through OpsCenter. The user is not able to view any of the data within the tables.

The user can take several actions through OpsCenter directed at the cluster or individual nodes. The user can change any of the configuration settings for the cluster to include the port, the bind address, authentication options, authorization options, and encryption options. The individual using OpsCenter can also delete the cluster. OpsCenter also enables the user to configure a node, stop a node, restart a node, decommission a node, and even add nodes. If authentication is enabled for Cassandra, but not OpsCenter, a malicious user can change Cassandra settings to remove the requirement for authentication.

There are several mitigations to this vulnerability. The first mitigation is to not install OpsCenter if it is not necessary for the database administrator to run the Cassandra cluster. The second mitigation is to change the bind address in the configuration file from all network interfaces to localhost. The third mitigation is to enable authentication for OpsCenter in the configuration file.



replica_placement_strategy org.apache.cassandra.locator.SimpleStrategy
replication_factor 1

employee

```
CREATE TABLE bankdb.employee (
  ssn text,
  lname text,
  password text,
  address text,
  dno int,
  fname text,
  minit text,
  salary float,
  sex text,
  superssn text,
  PRIMARY KEY (ssn, lname, password)
) WITH CLUSTERING ORDER BY (lname ASC, password ASC) AND
  bloom_filter_fp_chance=0.010000 AND
  caching='{"keys":"ALL", "rows_per_partition":"NONE"}' AND
  comment='' AND
  dclocal_read_repair_chance=0.100000 AND
  gc_grace_seconds=864000 AND
  read_repair_chance=0.000000 AND
  compaction={'class': 'SizeTieredCompactionStrategy'} AND
  compression={'sstable_compression': 'LZ4Compressor'};
```

Figure 5. Employee table CQL information available through OpsCenter.

THIS PAGE INTENTIONALLY LEFT BLANK

VII. ENCRYPTION

This chapter examines the vulnerabilities arising from encryption. We first provide a definition of encryption. We finish with examining MySQL, MongoDB, and Cassandra for any encryption vulnerabilities and provide mitigations to remedy these vulnerabilities.

A. DEFINITION OF ENCRYPTION

Vulnerabilities involving encryption can provide significant problems to organizations using any type of DBMS. "Missing encryption of sensitive data" is number eight on the 2011 CWE/SANS Top 25 list, and "sensitive data exposure" is number six on the 2013 OWASP Top 10 Most Critical Web Application Security Risks list [7][8].

We define encryption as a means of ensuring confidentiality of data [28]. In regards to databases there are two instances where the user must apply encryption in order to protect data. The first instance occurs when the user stores data in the database. This is referred to as "data at rest" [28]. If an attacker is able to gain access to the underlying system on which the DBMS is installed and read the data, then the data will be compromised if it is not encrypted. Encrypting all data in the database can add significant processing time for queries since the DBMS must decrypt the data in order to run a query against the data. A database administrator may choose to encrypt the most sensitive data in order to balance security with processing time.

The second instance occurs when data moves across the network from the presentation tier to the DBMS and vice versa. This is referred to as "data in transit" [28]. If an attacker is able to sniff data on the network, then the data will be compromised if it is not encrypted.

B. VULNERABILITIES AND MITIGATIONS

This section examines encryption vulnerabilities in MySQL, MongoDB, and Cassandra and presents mitigations against those vulnerabilities where applicable.

1. MySQL

MySQL does not enable encryption between client and server by default [16]. The database administrator can modify the configuration file to require encryption for all transactions between the client and server. This will ensure that data is protected in transit. In order to protect data at rest, MySQL offers built-in functions for encrypting data before storing the data in the database [16].

2. MongoDB

MongoDB does not enable encryption by default for client to server communication, nor does it enable encryption be default for node to node communication [21]. The database administrator can enable TLS/SSL encryption for both client to server and node to node communication by changing the configuration file. The HTTP interface for MongoDB does not support any encryption [19]. In order to offer an HTTP interface and protect data in transit, the database administrator could place a proxy server in front of MongoDB that does support encryption [19].

MongoDB does not offer any built-in functions for encrypting data prior to storage [19]. A database administrator wishing to encrypt data at rest must use a third party application in order to encrypt data. The use of a third party application can increase the complexities of query operations if the data to be queried is encrypted.

3. Cassandra

Cassandra does not enable encryption by default for client to server communication, node to node communication, or client to node communication [22]. In order to mitigate this, the database administrator can modify the configuration file to enable TLS/SSL encryption for client to server, node to node, and client to code communications. The HTTP interface for Cassandra, OpsCenter, does not enable encryption by default, however the database administrator can enable encryption through the OpsCenter menu [26].

Built-in functions do not exist in Cassandra to encrypt data for storage in the database [19]. As with MongoDB, a database administrator should use a third party application to protect sensitive data at rest through encryption.

THIS PAGE INTENTIONALLY LEFT BLANK

VIII. AUTHENTICATION AND AUTHORIZATION

This chapter examines the vulnerabilities arising from authentication and authorization. We first provide a definition of authentication and authorization. We finish with examining MySQL, MongoDB, and Cassandra for any authentication and authorization vulnerabilities and provide mitigations to remedy these vulnerabilities.

A. DEFINITION OF AUTHENTICATION AND AUTHORIZATION

Authentication and authorization are very important for properly securing a DBMS. "Missing authentication for critical function" ranks at number five and "missing authorization" ranks at number six on the 2011 CWE/SANS Top 25 list [8].

While they are distinct in definition, we include authentication and authorization together because they are very closely related. They are both concerned with different aspects of the user. Authentication focuses on the identity of a user while authorization focuses on the allowed accesses of a valid user [21].

We define authentication as the process by which a computer system confirms whether or not a user is who he or she claims to be [28]. Authentication is based on the factors of an individual of which there are currently three: something you have, something you know, and something you are [40]. An example of "something you have" is some type of object such as a phone or an identification card. An example of "something you know" could be a password or personal identification number. An example of "something you are" is a biological marker such as a fingerprint or DNA [40]. An authentication mechanism may use single-way factor, two-way factor (a combination of two of the three factors), or three-way factor authorization. In the case of single-way authentication an individual first presents credentials to the computer system usually in the form of a user id or name and a password. The computer system then checks if the user id exists in the user database, and if so, whether or not the given password matches the password in the database for the given user. Finally, the computer system either grants or denies access to the user based on the preceding check.

We define authorization as the apparatus by which a computer system grants a user access rights to resources, denies a user access rights to resources, or verifies the user's current access rights to resources [28]. A user in this case could be either an authenticated user or a guest or anonymous user.

B. VULNERABILITIES AND MITIGATIONS

This section examines authentication and authorization vulnerabilities in MySQL, MongoDB, and Cassandra and presents mitigations against those vulnerabilities where applicable.

1. MySQL

MySQL enables both authentication and authorization by default [16]. Additionally, when the database administrator creates a new user, the new user has limited privileges by default [16]. The database administrator should take care, however to review the user database. There is an installation option to create anonymous accounts which neither have a user name, nor do they require a password [16]. If this option was used at installation, then the database administrator should delete these accounts unless they are absolutely necessary. MySQL offers fine granularity for authorization. The database administrator can assign privileges based on MySQL defined user roles, or selectively grant privileges to users for an entire database or for specific objects within the database [16].

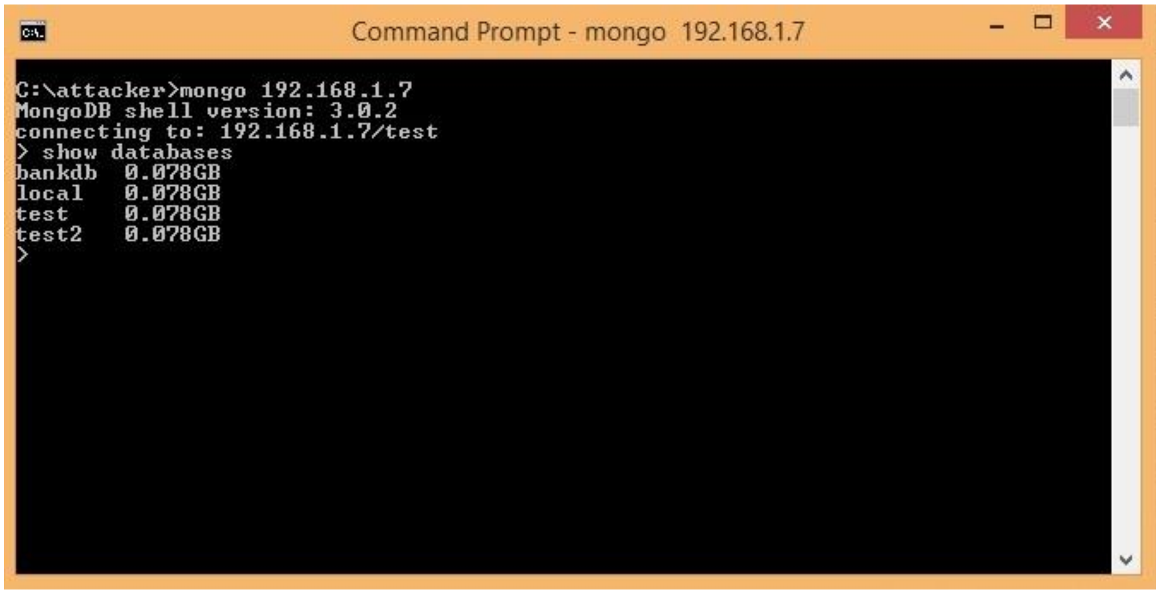
The database administrator should ensure that the presentation tier connects to MySQL with limited privileges. An attacker that is able to compromise the presentation tier through an injection or other means will operate at the privileges used to connect to MySQL. Stored procedures execute at the privilege level of the user who defined the stored procedure [16]. The database administrator should ensure that users who define stored procedures do not have excessive privileges in the event that an attacker compromises a stored procedure.

2. MongoDB

MongoDB does not enable authentication or authorization by default [21]. This is a significant vulnerability when combined with the network connection default of binding to all network interfaces. Anyone with a MongoDB client program can connect to any MongoDB database that is using all default configurations if the IP address of the system on which MongoDB is installed is known (see Figure 6).

The database administrator can change the configuration file to enable both authentication and authorization. When authentication is enabled, the default authentication method for MongoDB is SCRAM-SHA-1 [21]. MongoDB also supports X.509 certificate authentication, LDAP proxy authentication, and Kerberos authentication [21]. MongoDB offers role-based access control using a series of built-in roles and the option of creating user-defined roles [21]. The latter option allows the database administrator to control privileges at the collection level.

The HTTP interface for MongoDB does not support authentication or authorization even if enabled in the configuration file [19]. In order to mitigate this, the database administrator should place a proxy server in front of the HTTP interface that supports authentication and authorization [19].



```
C:\attacker>mongo 192.168.1.7
MongoDB shell version: 3.0.2
connecting to: 192.168.1.7/test
> show databases
bankdb  0.078GB
local   0.078GB
test    0.078GB
test2   0.078GB
>
```

Figure 6. Connecting to MongoDB with a Mongo client.

3. Cassandra

Cassandra does not enable authentication or authorization by default [22]. Cassandra is not as vulnerable as MongoDB using default configurations, however as the default network connection for Cassandra is to bind to localhost. The database administrator can enable both authentication and authorization in the configuration file. When authorization is enabled, the database administrator can grant privileges either using role-based access control or user-based access control [22].

The HTTP interface for Cassandra, OpsCenter, supports authentication and authorization, but this is not enabled by default even if authentication and authorization is enabled in the Cassandra configuration file [26]. In order to extend authentication and authorization to OpsCenter, the database administrator must enable these protections through the OpsCenter menu [26].

IX. CONCLUSION

This thesis focused on identifying security vulnerabilities in DBMS, both relational and NoSQL, and presenting mitigations against those vulnerabilities. We tested and examined MySQL, MongoDB, and Cassandra, which are all open source and representative of relational, NoSQL document, and NoSQL column-family DBMS respectively.

We identified five categories of security vulnerabilities that are common to relational and NoSQL DBMS and are significant according to the most recent OWASP and CWE/SANS vulnerability rankings. These vulnerabilities are injection, misconfigured databases, HTTP interfaces, encryption, and authentication and authorization. We also provided a methodology to examine MySQL, MongoDB, and Cassandra against each vulnerability.

We showed that presentation tiers for MySQL written in Python and Java can be vulnerable to several categories of injection: tautologies, illegal/logically incorrect queries, and union queries. We also showed that these vulnerabilities can be mitigated with parameterized queries. MySQL binds to all network interfaces by default, which is a security concern, however MySQL also enables authentication and authorization by default, which mitigates the vulnerability. MySQL does not enable encryption by default, be we showed that the database administrator can mitigate this vulnerability by modifying the configuration file to require encryption for data in transit, and utilize built-in functions to encrypt data at rest.

We showed that presentation tiers for MongoDB written in Python can be vulnerable to the tautology, illegal/logically incorrect query, and piggy-back query categories of injections. We also showed that these vulnerabilities can be mitigated by using safe functions for accepting user input such as “input()” in Python 3.X and “raw_input()” in Python 2.X. We showed that the default configurations for network connections and authentication and authorization present a significant vulnerability, and that this vulnerability can be mitigated by enabling authentication and authorization or changing network connection settings to bind to localhost. The HTTP interface for MongoDB is disabled by default, but we showed that this feature can present significant vulnerabilities when fully enabled. MongoDB neither enables encryption by default for data in transit, nor does it offer any built-in functions for encrypting data at rest. The former vulnerability can be mitigated by using supported encryption schemes and the latter can be mitigated with third party encryption applications.

We showed that presentation tiers for Cassandra written in Python and Java can be vulnerable to illegal/logically incorrect query injections. We also showed that this vulnerability can be mitigated by ensuring error messages are not returned to the user. Cassandra neither enables encryption by default for data in transit, nor does it offer built-in functions for encrypting data at rest. The general mitigations for MongoDB apply to Cassandra for this vulnerability. Cassandra's HTTP interface, OpsCenter, is not a default installation option, but can present vulnerabilities when used with default configuration options. We showed these vulnerabilities can be mitigated by changing the network connection to bind to localhost or enable authentication and authorization for OpsCenter. Cassandra does not enable authentication and authorization by default, but the database administrator can enable these protections to protect against this vulnerability.

There is room for future work in this area. This thesis only covered MySQL, MongoDB, and Cassandra, but there are many other popular open source DBMS. This thesis also did not consider the key-value or graph categories of NoSQL DBMS. In regards to the injection vulnerability, another area of future work is examining vulnerabilities in presentation tiers written in other languages besides Python and Java.

LIST OF REFERENCES

[1] L. Neal, "Will NoSQL databases live up to their promise?," *Computer*, no. 43–2, pp. 12–14, 2010.

[2] A. Moniruzzaman and S. Hossain, "NoSQL database: New era of databases for big data analytics—classification, characteristics and comparison," *International Journal of Database Theory and Application*, vol. 6, no. 4, pp. 1–14, 2013.

[3] D. Harris. (2013, Jun. 7). Under the covers of the NSA's big data effort. [Online]. Available: https://gigaom.com/2013/06/07/under-the-covers-of-the-nsas-big-data-effort/

[4] D. Baum, "The Department of Defense (DOD) and open source software," Oracle, Redwood Shores, CA, Sep. 2013.

[5] DOD IG - Digital Strategy. (2015). DODig.mil. [Online]. Available: http://www.DODig.mil/digitalstrategy/

[6] A. Corrin. (2013, Nov. 19). Has open source officially taken off at DOD? *Federal Computer Week* [Online]. Available: https://fcw.com/Articles/2013/11/19/DOD-open-source.aspx

[7] "OWASP Top 10 - 2013," Open Web Application Security Project, 2013.

[8] CWE -2011 CWE/SANS Top 25 Most Dangerous Software Errors. (2011). The Mitre Corporation. [Online]. Available: http://cwe.mitre.org/top25/

[9] M. Lennon. (2011, Apr. 2). Massive breach at Epsilon compromises customer lists of major brands. [Online]. Available: http://www.securityweek.com/massive-breach-epsilon-compromises-customer-lists-major-brands

[10] J. McEntegart. (2011, Apr. 27). Sony confirms massive data breach affecting 70 million PlayStation Network users. [Online]. Available: http://www.tomsguide.com/us/Sony-PSN-PSN-Data-Breach-Loss-Credit-Card-Fraud,news-10982.html

[11] B. Hardekopf. (2014, Oct. 3). Major data breach at JP Morgan Chase hits 76 million households. [Online]. Available: http://www.lowcards.com/major-data-breach-jp-morgan-chase-hits-76-million-households-27953

[12] M. Virtanen. (2014, Jul. 15). 22.8 million personal records of New Yorkers exposed. *Rochester Democrat and Chronicle* [Online]. Available: http://www.democratandchronicle.com/story/news/2014/07/15/data-security-breaches-new-york/12706213/

[13] J. Oldshue. (2014, Dec. 9). Sony data breach worse than expected. [Online]. Available: http://www.lowcards.com/sony-data-breach-goldmine-identity-thieves-29386

[14] J. Scharr. (2015, Feb. 5). Anthem Health Care data breach: What to do now. [Online]. Available: http://www.tomsguide.com/us/anthem-data-breach,news-20411.html

[15] L. Shane III. (2015, Jul. 13). OPM data breach to be subject of hearings. *MilitaryTimes* [Online]. Available: http://www.militarytimes.com/story/military/capitol-hill/2015/07/13/hasc-opm-data-breach/30080025/

[16] *MySQL 5.6 Reference Manual*, Oracle, 2015, pp. 4-3016.

[17] C. Anley, "Advanced SQL injection in SQL server applications," Next Generation Security Software Ltd., 2002.

[18] D. Harris. (2013, Aug. 27). 10gen embraces what it created, becomes MongoDB Inc. [Online]. Available: https://gigaom.com/2013/08/27/10gen-embraces-what-it-created-becomes-mongodb-inc/

[19] L. Okman, N. Gal-Oz, Y. Gonen, E. Gudes and J. Abramov, "Security issues in NoSQL databases," in *Trust, Security and Privacy in Computing and Communications (TrustCom), 2011 IEEE 10th International Conference on*, 2011, pp. 541–547.

[20] A. Floratou, N. Teletia, D. DeWitt, J. Patel and D. Zhang, "Can the elephants handle the NoSQL onslaught?," *Proc. VLDB Endow*., vol. 5, no. 12, pp. 1712–1723, 2012."

[21] *MongoDB Documentation Release 3.0.5*, MongoDB Inc., 2015, pp. 3-356.

[22] *Apache Cassandra 2.2 for Windows Documentation*, Datastax Inc., 2015, pp. 8-97.

[23] J. Han, E. Haihong, G. Le and J. Du, "Survey on NoSQL database," in *Pervasive Computing and Applications (ICPCA), 2011 6th International Conference* on, 2011, pp. 363–366.

[24] The Apache Cassandra project. Apache Software Foundation. [Online]. Available: http://cassandra.apache.org/download/. Accessed Jul. 13, 2015.

[25] *CQL for Cassandra 2.x Documentation*, Datastax Inc., 2015, pp. 6-28.

[26] *OpsCenter 5.2 User Guide Documentation*, Datastax Inc., 2015, pp. 8-78.

[27] W. Halfond, J. Viegas and A. Orso, "A classification of SQL injection attacks and countermeasures," in *Proceedings of the IEEE International Symposium on Secure Software Engineering*, 2006.

[28] A. Zahid, R. Masood and M. Shibli, "Security of sharded NoSQL databases: A comparative analysis," in *Information Assurance and Cyber Security (CIACS), 2014 Conference on*, 2014, pp. 1–8.

[29] G. Buehrer, B. Weide and P. Sivilotti, "Using parse tree validation to prevent SQL injection attacks," in *Proceedings of the 5th International Workshop on Software Engineering and Middleware*, 2005, pp. 106–113.

[30] S. Kerner. (2013, Nov. 25). How was SQL injection discovered? [Online]. Available: http://www.esecurityplanet.com/network-security/how-was-sql-injection-discovered.html

[31] S. Son, K. McKinley and V. Shmatikov, "Diglossia: Detecting code injection attacks with precision and efficiency," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, 2013, pp. 1181–1192.

[32] R. Chandrashekhar, M. Mardithaya, S. Thilagam and D. Saha, "SQL injection attack mechanisms and prevention techniques," in *Advanced Computing, Networking and Security*, Springer Berlin Heidelberg, 2012, pp. 524–533.

[33] M. Muthuprasanna, K. Wei and S. Kothari, "Eliminating SQL injection attacks – a transparent defense mechanism," in *Website Evolution, 2006. WSE'06. Eighth IEEE International Symposium on*, 2006, pp. 22–32.

[34] W. Halfond and A. Orso, "Combining static analysis and runtime monitoring to counter SQL-injection attacks," *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 4, p. 1, 2005.

[35] W. Halfond, A. Orso and P. Manolios, "Using positive tainting and syntax-aware evaluation to counter SQL injection attacks," in *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2006, pp. 175–185.

[36] R. McClure and I. Kruger, "SQL DOM: Compile time checking of dynamic SQL statements," in *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, 2005, pp. 88–96.

[37] Java - basic datatypes. [Online]. Available: http://www.tutorialspoint.com/java/java_basic_datatypes.htm. Accessed Sep. 30, 2015.

[38] J. Heyens, K. Greshake and E. Petryka, "MongoDB databases at risk," Center for IT-Security, Privacy, and Accountability, Jan. 2015.

[39] A. Ron, A. Shulman-Peleg and E. Bronshtein, “No SQL, no injection?.”

[40] Van Tilborg, C. A. Henk , and Sushil Jajodia, *Encyclopedia Of Cryptography And Security*, New York: Springer, 2011.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California